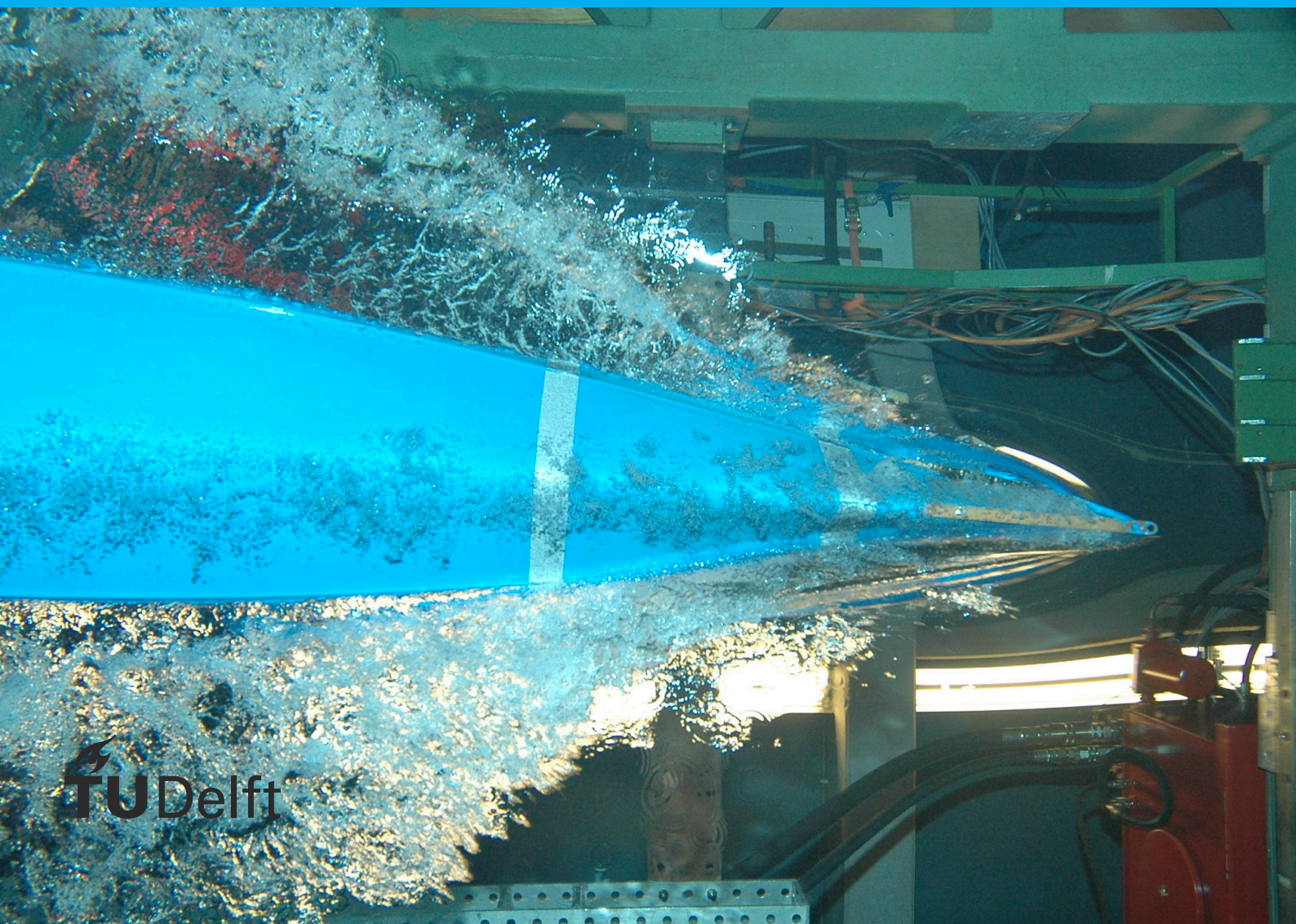
Embedded Memory Security

Securing the IoT from the hardware level

Hans Okkerman

Preventing hardware based attacks on the memory of IoT devices



Embedded Memory Security

Securing the IoT from the hardware level

by

Hans Okkerman

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on November 30 at 14:00.

Student number: 4290453
Project duration: September 1, 2019 – November 30, 2020
Thesis committee:

Prof. dr. ir S. Hamdioui	TU Delft
Dr. ir. M. Taouill	TU Delft, supervisor
Dr. ir. R. van Leuken	TU Delft
Dr. C. Reinbrecht	TU Delft

TU Delft

Abstract

The Internet of Things (IoT) has grown dramatically over the past years. Largely autonomous, lightweight devices with an internet connection have been integrated into many aspects of daily life; from consumer products to industrial processes and from medical applications to critical infrastructures. Cisco predicts that by 2023, more than half of the internet connections will belong to IoT devices. With the impact of cybercrime ever growing, currently leading to a loss of 600 billion US dollar per year according to McAfee, the Internet of Things has become an attractive target. Successful attacks have already been demonstrated on smart cars, home security cameras, heart defibrillators, the Ukrainian power grid and military drones, just to name a few. Attacks of this nature are expected to intensify as more 'things' are connected to the internet, either by criminals looking for quick money, companies sabotaging competitors or countries waging cyber warfare. This demonstrates the need for strong security for IoT devices.

Attacks on IoT devices can come from three distinct direction: The network, the software or the hardware level. Where network protocols and software applications can be updated when issues are found, this is not the case for hardware, which must be designed to be secure from the start. Furthermore, the way IoT devices are installed in the field makes hardware based attacks particularly relevant. Examples include the probing of traces and pins, fault injections to cause unintended behaviour, modifications of the firmware, side-channel analysis, stealing of data etcetera. Countering many of these attacks requires integrity verification of the attached memory chips of a device, to make sure that the applications have not been tampered with. Existing security measures implemented in high performance processors, such as Intel SGX and more recently AMD SEV, can encrypt and secure the memory of a system. These implementations however are not available for the lightweight microcontrollers and processors generally found in IoT devices. ARM TrustZone is a common security solution found in embedded devices to provide protection against untrusted software, but does not defend against hardware tampering. As such, a lightweight solution is required aimed at the constrained environments presented by IoT devices, to ensure the integrity of external memory modules.

This thesis presents the Embedded Memory Security (EMS) module as a way to ensure the integrity and authenticity of data and applications, as well as its confidentiality if required. It is targeted at lightweight systems with a small hardware budget. The module sits on-die with the central processor of the device and secures all data being transferred between it and external memory. Integrity is verified through Message Authentication Codes (MAC) generated with SipHash for each memory transfer. The lightweight block cipher Prince is used to provide confidentiality through encryption. Five variants of this module with different levels of security and optimizations are developed and integrated into a processor development platform. Benchmark runs showed that under realistic cache conditions, their impact was limited to a 25% increase in execution time at worst. Three attacks were performed on the platform with the modules, indicating that they protect against several types of hardware attacks on memory. Finally, the hardware cost in area requirements was determined and found to be less than half of the microcontroller-class RI5CY core, excluding its caches, and only 3% of the Linux-capable Ariane core. In addition to the EMS modules, two security extensions are proposed that utilize the modules to provide a secure method of updating devices.

Contents

List of Figures

List of Tables

Acronyms

AES Advanced Encryption Standard.

CCM Cipher block Chaining Mode.
CVE Common Vulnerabilities and Exposures.

DDoS Distributed Denial of Service.
DoS Denial of Service.

ECB Electronic CodeBook.
ECU Engine Control Unit.
EM Electro Magnetic (radiation).
EMS Embedded Memory Security.

FPGA Field Programmable Gate Array.

GCM Galois/Counter Mode.
GE Gate Equivalents.

I2C Inter-integrated circuit.
IC Integrated circuit.
IoT Internet of Things.
IP Internet Protocol.

JTAG Joint Test Action Group.

KDF Key Derivation Function.

LUT LookUp Table.

MAC Message Authentication Code.
MitM Man in the Middle.
MMU Memory Management Unit.

NAS Network-Attached Storage.
NSA National Security Agency.

OCB Offset CodeBook.
OS Operating System.

PCB Printed Circuit Board.
PLC Programmable Logic Controller.
PUF Physical Unclonable Function.

RAM Random Access Memory.
RFID Radio-Frequency IDentification.
RTSO Real-Time Operating System.

SCA Side-Channel Analysis.
SGX Software Guard Extensions.

SoC System on Chip.
SPI Serial Peripheral Interface.
SPN Substitution Permutation Network.

TPM Trusted Platform Module.

UART Universal Asynchronous Receiver-Transmitter.

1 Introduction

This chapter given an introduction to the topics discussed in this thesis. It first provides the motivation by going over the increasing costs of cybercrime, relates it to the growth of the Internet of Things and highlights the need for strong protections against hardware-based attacks in Section 1.1. Next, Section 1.2 covers existing security measures and points out the areas that are lacking. Section 1.3 then summarizes the contributions of this thesis and finally, Section 1.4 presents the outline of the remaining chapters of this thesis.

1.1. Motivation

With the integration of 'smart' functionality to everyday items, more and more devices are connected to the internet. Utilities that were once stand-alone units such as home thermostats, cars and refrigerators can now offer additional services to automate the daily lives of their owners. Similarly, in order to optimize processes in industrial and commercial settings, internet-connected controllers, security cameras and even wireless sensor networks have become commonplace. Together, such devices make up what is known as the Internet of Things (IoT); an ever growing collection of largely autonomous devices with an internet connection. As with most things, there are those who wish to abuse this new technology for their own gain at the cost of others: Criminals infecting many devices to form botnets, corporate espionage or sabotage of processes of competing companies and even full-scale cyber warfare between rivalling nations; all highlight the need for strong security measures.

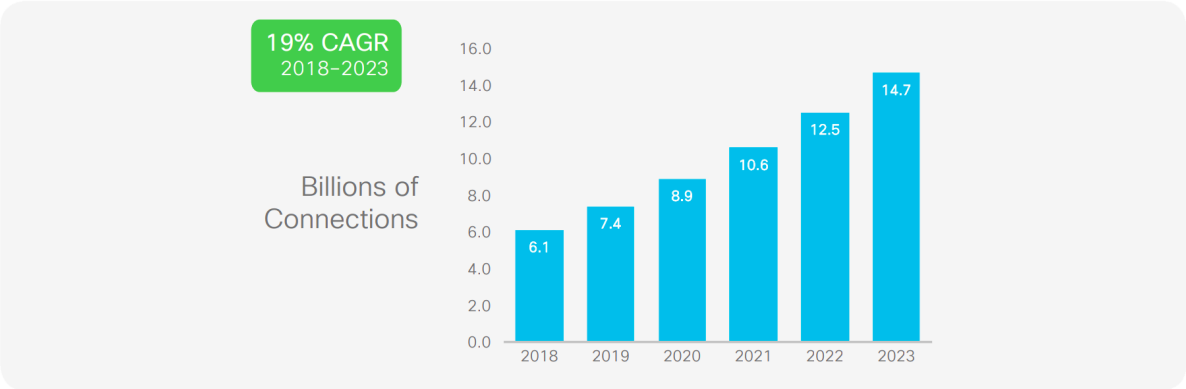


Figure 1.1: Cisco's predictions of current and future growth of the IoT [1]

The growth of the IoT is expected to continue rapidly. Cisco estimates that the amount of connected IoT devices will reach a total of 14.7 billion by 2023, 2.4 times as many as in 2018, and making up half of the total internet connections [1]. This growth is illustrated in Figure 1.1. In particular, the amount of internet-connected cars is expected to increase by 30% annually, with city and energy production applications following at 26% and 24% respectively. These trends are shown in Figure 1.2. Along with

the growth of the internet, so grows the cost inflicted by cybercrime. In 2014, McAfee reported a global annual loss of 500 billion US dollars caused by this type of crime. In 2018 this figure was increased by 20% to 600 billion dollars, making up 0.8% of the global GDP [2]. This indicates that combating cybercrime should be a top priority.

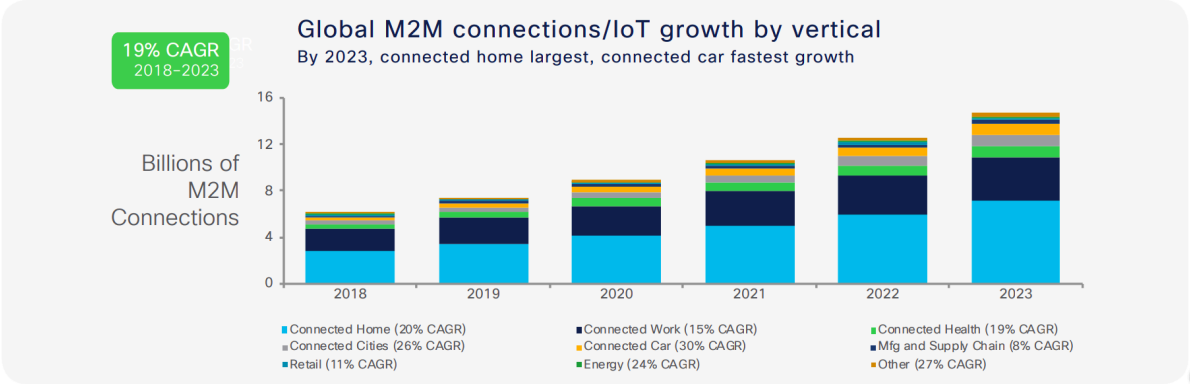


Figure 1.2: Cisco's expected application makeup of the IoT market [1]

Several attacks specifically targeted at IoT devices have already been demonstrated: Stuxnet for example, caught the world by surprise when it infected computers all over the planet [3], while targeting specific controllers to destroy Iran's nuclear centrifuges. In 2016, the Mirai worm rapidly spread and infected hundreds of thousands of IoT devices through weak default login credentials. These then became part of a large botnet, responsible for some of the heaviest DDoS attacks on record [4]. Aside from such large-scale attacks, some have been shown to have much more personal consequences. For example, researchers were able to take full control over a Jeep Cherokee car through the internet. No physical tampering was required due to an insecure multimedia system with an internet connection [5]. This enabled them to change the radio volume, toggle the windscreen wipers as well as disable the brakes, transmission and the engine itself while driving. Similarly, Hanna et al. could intercept data in a defibrillator and install custom updates [6]. If performed outside of a research setting, this could pose a real threat to heart patients. Finally, in 2015, attackers managed to breach systems of the Ukrainian power grid, showing the real-life consequences of cyber-warfware when 230.000 people were left without power for several hours.

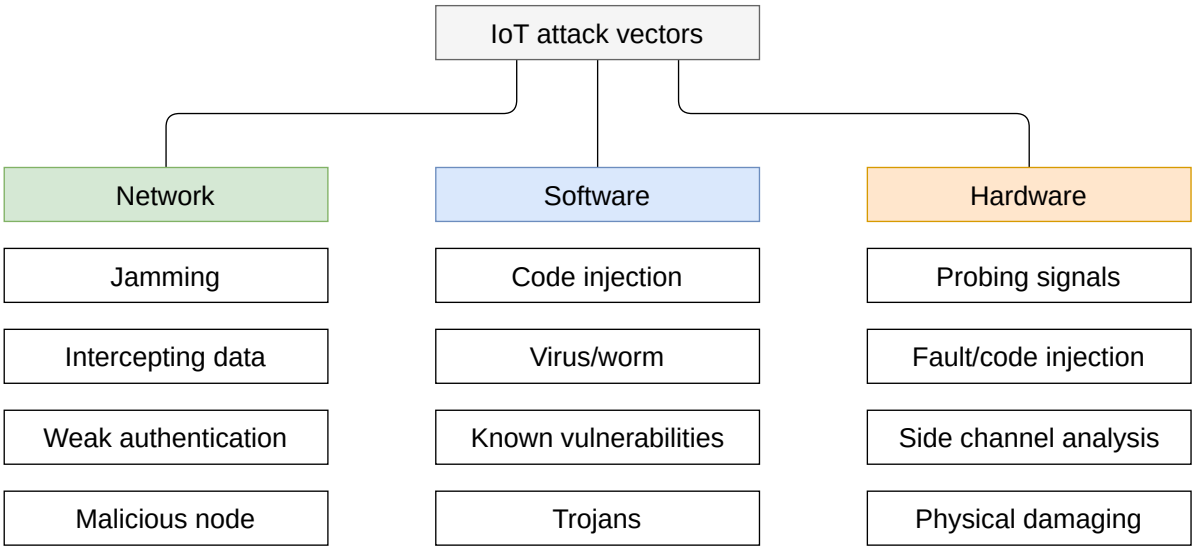


Figure 1.3: Categories of attack vectors on IoT devices, with non-exhaustive list of examples

The above makes clear that there is not just one type of IoT device with one type of attack. In

fact, there are many different types of attacks that take widely varying approaches. Based on their primary attack vector, these can be grouped into three main categories: Network based, software based and hardware based. An overview of these categories with some example attacks for each, is shown in Figure 1.3. This varied nature of the attack types makes that there is no 'one-size-fits-all' solution to protect against all of them. Each type of attack requires specific countermeasures. Much research has been performed on developing secure software and network solutions already, but the hardware security is lacking. Furthermore, weaknesses in software can be fixed through updates, whereas hardware cannot simply be updated once deployed. Finally, since IoT devices are typically installed 'in the field' unlike servers and personal computers, hardware based attacks are particularly relevant. Fault injections can be performed to extract secret keys or bypass secure boot setups [7] [8]. Malicious firmware could be flashed to their memory through physical access [9], and data could be falsified to report wrong values and manipulate several processes.

This thesis focusses on protecting IoT devices from hardware based attacks on their memory modules. A large number of techniques to physically modify the data stored in memory are available and easy to perform. Defending against these attacks by ensuring the integrity and authenticity of data creates a foundation of trust, upon which additional measures can be constructed against remaining threats.

1.2. State of the Art

The security of computer systems has long been a topic of research. Especially in server and desktop applications, the advances in computing power have allowed for extensive security frameworks to be developed. Intel's Software Guard Extensions (SGX) for example, enables applications to encrypt parts of the memory they are using [10]. Amongst other things, this protects them from hardware-based attacks on memory. It does however require applications to be specifically written to make us of it, and is not available for low power chips and embedded devices. Similarly, [11] presents a way to encrypt sections in RAM through software when no hardware support for this is available. It does however come with a severe performance impact and does not protect against hardware attacks, since it stores its encryption key in plaintext in the same RAM.

ARM TrustZone, on the other hand, is a technology that is implemented on several ARM architectures, specifically those aimed at mobile and embedded devices [12]. It sets up two environments; a secure and a non-secure world, on the same chip. TrustZone provides hardware barriers that enable applications from the secure world to access all resources of the device, while the secure resources are shielded from the non-secure world. This way, unauthorized reads or writes to sensitive data in memory by non-secure applications can be prevented. TrustZone is however not intended as a measure against physical hardware tampering and does not protect against any such attacks.

Several methods of remote attestation exist, which allow a remote 'base station' to verify the integrity of its nodes. The base station regularly sends a verification request to its nodes, which then perform a verification action and respond. This action could be to unseal and use data stored in a Trusted Platform Module (TPM) to generate a response, which can only happen when the device is in an identical state as when it was sealed [13]. Similarly, self-checksumming code could be used as is the case for SCUBA [14]. Here, the base station orders the execution of a self-verifying application, which then calculates a hash over memory contents. At the same time, the base station knows the supposed state of the nodes and performs the same operations. If the reply of the node matches with what it should be, the node can be considered verified. Such a setup does however require a significant amount of network communications and processing power at both the node and base station side, especially as the amount of nodes increases. Furthermore, there are attacks that may counter such a setup [14]. Finally, remote attestation attempts to detect an attack, after it has taken place. It does nothing however to prevent one from happening in the first place.

Some proposals have been made that aim to prevent attacks on memory, through security modules that provide encryption and integrity verification. Suh et al. developed a module in 2003 that uses AES to encrypt data and implements a hash-tree with SHA-1 [15]. The design includes timestamps that are either stored on-chip or in memory, in order to also protect against replay attacks. In 2006, Elbaz et al. developed a module with the same goals [16]. Their design uses randomly generated tags that are stored on-chip for each memory address, and generates new keys for each running application. When data is encrypted using AES to be written to memory, this tag is combined with the plaintext.

As data is read back from memory and decrypted, this tag can be compared to the one stored on-chip to verify integrity without a dedicated MAC function. Finally, in 2013, Crenne et al. devised an elaborate setup using AES in Galois/Counter Modes (GCM) [17], both encrypting and authenticating data stored in memory. In this mode for each write to memory, a new keystream must be generated. This is done here by producing and storing time-stamps for each memory address on chip, keeping ID's of memory segments and also including their address itself. The proposed scheme is shown in Figure 1.4.

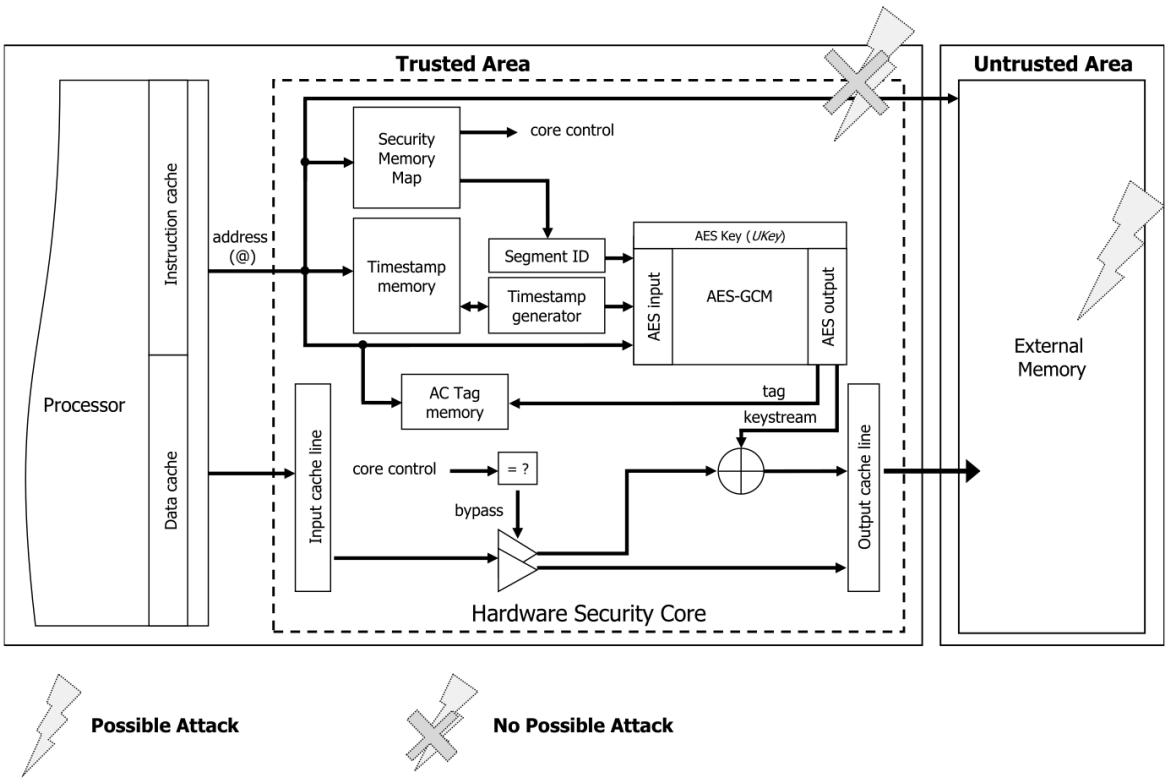


Figure 1.4: Write operation with AES in GCM mode using timestamps, addresses and segment ID's to secure external memory [17]

Though offering protection against hardware attacks on memory, none of these three implementations can be called particularly lightweight regarding hardware requirements: AES was never intended to be so, neither was SHA-1, nor is storing 64 bits of data for each address in memory on-chip. In the heavily power and area constrained environments of IoT devices, the cost of these modules underlines the need for a lightweight alternative.

1.3. Contribution

The main goal of this thesis is to understand the types of threats that IoT devices face, and to develop a security solution against physical tampering. First, known attacks and existing countermeasures were explored in detail to find what areas need addressing. Next, a hardware module was proposed, developed and tested to physically protect the external memory modules of a device. Finally, security extensions were proposed that can be added to the module in order to also provide a secure method of transferring and installing updates. This thesis's main contributions are listed as follows:

- **Proposal of Embedded Memory Security (EMS) module:** It is found that IoT platforms are particularly attractive targets for and vulnerable to hardware tampering. Their external memory modules are especially exposed to fault and code injection, malicious firmware, as well as probing. To counter attacks targeted at such memory chips, a security module is presented that ensures the integrity, authenticity and optionally the confidentiality of stored code and data. This module is intended to serve as a base for further extensions to counter other types of attack.

- **Proposal of Security Extensions for EMS module:** Two security extensions are proposed to secure the transfer and installation of updates to IoT devices. The first proposes the use of a unique device ID provide a custom, authenticated update, and the second proposes a method to bind this new software to the specific hardware it is installed on. This is intended to prevent attackers from forging malicious updates for other devices, even in the event they manage to completely compromise one of them.
- **Evaluation on Performance, Security and Hardware Overhead:** Five variants of the presented EMS module were implemented in Verilog. They were then added to the RAM interface of a processor platform and both simulated and synthesized to an FPGA. Their performance impacts were tested by running several benchmarks with multiple cache sizes and by measuring their hardware area requirements. Their provided security was determined by performing attacks on the platform.
- **Comparison with state-of-the-art:** Two state-of-the-art solutions that target the same objective of EMS module are compared in terms of timing and hardware overhead.
- **Survey on relevant attacks for IoT devices:** A collection of attacks are grouped into one of three categories, based on the main attack vector being network, software or hardware based. In each category, four different attack types are discussed. The workings of each type is briefly covered, some examples are provided and potential countermeasures are mentioned. This gives an overview of the current threats to IoT security.

1.4. Thesis Organisation

The remainder of this report consists of 6 more chapters. The first two provide an overview of the threats that IoT devices face in the form of different types of attacks, and the risk that these devices pose to their surrounding when an attack is successful. The next two present the proposed Embedded Memory Security modules, as well as their implementation and design choices. The last two chapters present the results and conclusions of this thesis. A more detailed description of the chapters is as follows:

Chapter 2 starts out with an overview of the Internet of Things, including common applications, device hardware and typical software. Next, it discusses some relevant security goals and requirements. Finally, three application categories are defined based on the severity of the consequences when a device in a particular application is compromised.

Chapter 3 covers three categories of attacks that can be performed on IoT devices, based on their primary attack vector; network, software or hardware based. It then explains the operation, gives examples and potential countermeasures to four relevant ones per category.

Chapter 4 presents the solution developed during this thesis. It first discusses the concepts of the developed EMS modules, followed by their operations, security considerations and possible extensions.

Chapter 5 covers the implementation of the EMS modules, as well as their design choices. Here, the selection process of the used development platform are discussed, as well as the cipher and MAC functions. It concludes with the setup and variants of the modules themselves.

Chapter 6 presents the results of this thesis. The developed modules are installed in the processor platform and synthesized to FPGA hardware, as well as simulated while running several benchmarks. Some attacks are performed against the platform with and without modules, and their hardware costs are compared to state-of-the-art solutions.

Chapter 7, finally, provides a summary of this thesis. The conclusions of the project are stated, finishing off with potential future work.

2

Introduction to the Internet of Things

The Internet of Things (IoT)) is a rapidly growing collection of electronic devices that are connected to the internet. These devices range from low powered sensor nodes, to full-fledged computers running a form of Linux. Similarly, their responsibilities vary from innocuous home thermostats controlling the temperature inside, to controlling parts of a nuclear powerplant. Based on their application, they may be an attractive target to cyber criminals in order to steal information, breach privacy, sabotage processes or simply to cause chaos. This chapter first given an overview of the Internet of Things, including types of applications, commonly used hardware as well as software in Section 2.1. Next, Section 2.2 explains four relevant security goals; Confidentiality, integrity, availability and authenticity of data. Finally, Section 2.3 groups IoT applications into three categories, based on the severity of the consequences when a device is compromised.

2.1. Internet of Things

This section will cover the basics of the Internet of Things. First, some typical applications are mentioned, followed by the levels of hardware performance that can be found in such devices. Finally, a list of software options is provided.

2.1.1. Applications

The application space of the IoT has become rather broad over the years and continues to expand. To give a quick overview of where IoT devices are being used, some categories and examples are briefly mentioned next:

- A wide variety of personal consumer electronics is available to make daily life easier and more connected. Examples include home automation devices such as 'smart' fridges and thermostats, as well as connected and autonomous cars.
- Corporate and industrial applications include sensor networks and controllers to measure many aspects of processes and their environments. These are then used to automate and optimize production.
- Critical infrastructures depend on IoT devices in the form of cameras and electronic signs to automate traffic control. Similar to the industrial setting, sensors are used in water treatment facilities and power plants as well.
- Medical uses include defibrillators which can remotely report their (battery) status, as well as remote surgery machines which allow surgeons to operate on far-away patients.
- Security devices are common in the form of smart-cards, PIN terminals and cameras. These are connected to the internet to, for example, verify authentications with some database.
- Military applications are expanding with wearable sensors and drones. These can be used to record and share battlefield information with the command-centre and units on the ground to gain a strategic advantage.

Though this list is far from exhaustive, it should make the point that IoT devices are being integrated

into a very broad and ever increasing range of applications.

2.1.2. Hardware

The devices mentioned above perform many different tasks in various environments. As such, their hardware requirements vary widely as well. Having less processing power available than required leads to problems, but having more than is needed is a waste of money, resources and energy. For clarity, most devices can be grouped in one of three categories based on their hardware performance, similar to other works [18]; low-end, medium-end and high-end:

- The lower end of the processing power spectrum contains devices such as wireless sensor nodes. These don't need a lot of computational performance to collect and occasionally transmit data. As they are often powered by batteries because of limitations in their installation locations, there are strong constraints on their energy consumption. Figure 2.1 shows an example of one such sensor node: A Tmote Sky MTM-CM5000-MSP. These lightweight devices generally contain small microcontrollers such as the 8-bit Microchip ATmega128 or the 16-bit Texas Instruments MSP430 as in the Tmote node; The latter running at around $8MHz$, coming with 10KB of RAM and 48KB of Flash, whereas the former is even more limited [19].

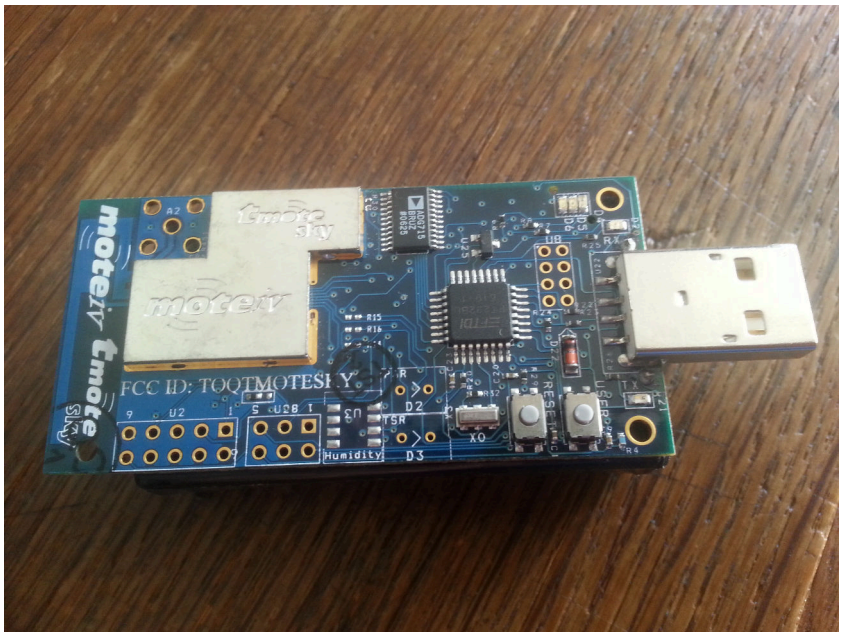


Figure 2.1: Tmote Sky MTM-CM5000-MSP battery powered wireless sensor node, using a TI MSP430 microcontroller

- The medium level is taken up by devices that require a bit more processing power to function. This power could be needed control displays and other user interfaces, process a larger stream of data such as audio or otherwise perform more compute intensive operations. Home automation as well as industrial PLC's would be some example applications where these are relevant. A more powerful microcontroller such as an STMicroelectronics STM32F4 and many others based around a 32-bit ARM core would be typical in such devices. These run at up to $180MHz$ and come with up to 256KB of onboard ram and 2MB of flash, as well as additional interfaces and accelerators [18].
- At the higher end of the processing power spectrum, one would find devices with fast internet connections, video processing capabilities and/or large data storages. Examples of such applications would be the multimedia systems in modern cars [20], IP cameras as well as the central base station of home automation [9] or sensor networks [18]. Instead of being based around a microcontroller, these instead step up to microprocessors with much higher performance and large off-chip memories. To name one, the Nest Thermostat shown in Figure 2.2, contains a Texas Instruments Sitara AM3703 processor with an Arm Cortex-A8 core. It runs at $1GHz$, and comes with 64MB of external RAM and 256MB of flash [9].

Figure 2.2: The Nest Thermostat, containing a higher-end microprocessor [9]

What kind of hardware will be used in a design is something that should be decided upon at early stages in development. Once a designer knows the required functionality of the device to be developed, this leads to a trade-off between required processing power, unit cost, ease of development and energy consumption. Software requirements follow a similar decision process, which will be covered next.

2.1.3. Software

Just like the hardware, the software running on IoT devices is not set in stone either. Though the hardware capabilities largely determine what kind of tasks it will be able to run, selecting suitable software for the application greatly influences the time it takes to develop as well as its performance. A trade-off will have to be made regarding code size, execution speed, real-time requirements, security, etcetera. As with hardware, the options can be split in three categories:

- The most lightweight option is to run applications bare-metal without an operating system. This works well when memory and processing performance are severely limited as is the case in smaller microcontrollers. By not having an operating system, all system resources are directly available to the applications with minimal overhead. One downside of this is that applications are less portable and have to be manually written and optimized for each different microcontroller. With multiple applications, determining which one gets to run when also becomes a task for the programmer. Finally, complicated network stacks and software security measures might be too large to fit the device's memory. Bare-metal applications are therefore mostly relevant at the lower end of the power spectrum, such as wireless sensor nodes [18].
- Lightweight operating systems are a step up in capabilities and ease of development, when multiple tasks need to be performed and scheduled. They make it easier to develop applications which can then run on other devices supported by the OS, with minimal changes to the code. Furthermore, they may provide a full networking stack and other functionality. One such lightweight OS is RIOT [21]. Real-Time Operating Systems (RTSO) are also an option at this level. They are intended to guarantee that tasks are completed before certain deadlines, which makes them ideal for timing-critical applications such as industrial controllers. An example of one such operating system is FreeRTOS.
- For the high-performance devices, a 'full' operating system would be the best option. Linux, android and Windows 10 IoT Core for example provide well known environments to develop and run applications. They come with fast networking stacks, full multitasking, enable graphical processing and support a wide range of high performance devices. Applications that could benefit from these functionalities are IP-cameras and devices with advanced graphical interfaces, such as the entertainment system in smart cars [20].

As before, choosing which way to go is something that happens in the early phases of development and is closely tied to the choice of hardware.

2.2. Security Goals

Internet of Things devices, just like other computer systems, deal with software and data. They perform measurements using sensors, run software stored in memory and transmit data to base stations and other systems. Depending on the application, data or algorithms may be proprietary or otherwise sensitive and should be prevented from falling into the wrong hands. Similarly, when it is critical that a program runs as intended and has not been tampered with, this will need to be verified. Finally, when systems depend on a constant stream of data whenever they need it, this too needs to be ensured.

The ISO/IEC 27000-series of standards deals with information security. It provides a vocabulary as well as recommendations on managing the security of code and data [22]. The standard defines information security as ensuring the 'CIA' triad, consisting of the concepts Confidentiality, Integrity and Availability of information. Many security papers follow the same setup [23] [24]. These three concepts are explained next:

- **Confidentiality** is defined as the 'property that information is not made available or disclosed to unauthorized individuals, entities, or processes' by the standard [22]. In the context of the Internet of Things, it can prevent adversaries from reading the contents of memory or communication. Encryption is a typical method to ensure confidentiality, where only those with knowledge the correct key have access to the information.
- **Integrity** is defined as the 'property of accuracy and completeness' [22]. In the context of the IoT, verifying integrity ensures that data has not been changed either accidentally through hardware or transmission errors, or on purpose by an adversary. Hash functions are a typical method to verify the integrity of data.
- **Availability** is defined as the 'property of being accessible and usable on demand by an authorized entity' [22]. Again, in the IoT context, ensuring availability means that data provided by a device can be accessed whenever needed. Increasing the availability can be done through robust communication channels and redundancy.

Authenticity is often regarded as an addition to these three, being related to Integrity:

- **Authenticity** is defined as the 'property that an entity is what it claims to be' by the standard [22]. Verifying authenticity in the IoT ensures that the origins of data are known and correct. Digital signatures and Message Authentication Codes (MAC) are used to guarantee the authenticity of data. An adversary may be able to generate data with a correct hash to pass integrity checks, but the authenticity verification will detect that it was not generated by a valid source.

Availability is mostly a concern for the communication of data. It is less relevant to data stored in memory on the device as, unless the device is broken, the memory chip and its contents are present and available. Integrity, authenticity and confidentiality on the other hand, are relevant both to communication and local storage. The first two guarantee that data has not been altered and tampered with, while the second ensures that it remains hidden from unauthorized entities. Implementing all three will make it very difficult for an adversary to change and data of his choosing, without being detected.

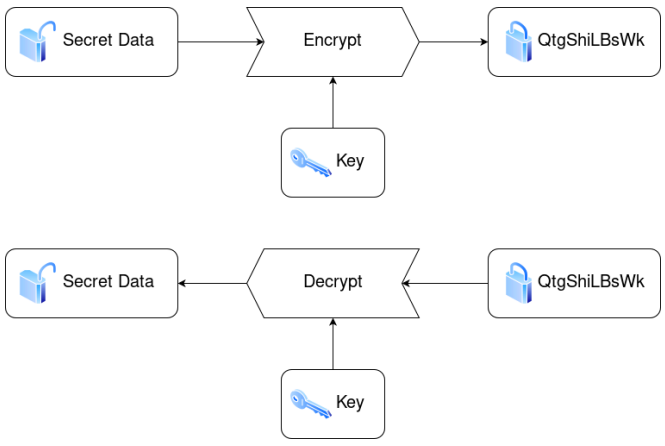


Figure 2.3: Basic concept of encryption. Only those with the correct key can recover the original data.

As mentioned above, encryption is widely used in computer systems to protect sensitive information against prying eyes and ensure confidentiality. It works by performing certain mathematical operations on data based on a secret key. Ideally, after encryption the result looks like random noise with no discernible information of the original plaintext. Any change, even a single bit, in the plain or ciphertext should lead to a completely different corresponding cipher or plaintext. Only those with knowledge of the secret key will be able to undo the transformation and get access to the original data. The basic concept is visualized in Figure 2.3. For a good introduction to cryptography in general the reader is referred to [25].

Figure 2.4: Basic concept of Message Authentication Codes (MACs). Small changes in input or key result in different hash.

Similarly, MACs are a type of digital signature used to verify both integrity and authenticity of data, and ensure that it has not been tampered with. They are based around a hash function with the addition of a secret key. A hash function takes data of arbitrary length as its input and generates a fixed-length output through some mathematical algorithm. Inverting the operations, i.e. finding a message that when hashed would lead to a given hash value must be unfeasible, making hashing a one-way function. Furthermore, any change in the input data should lead to a completely different hash value, thus allowing any tampering to be detected and integrity to be verified by comparing the hashes [25]. Though this would be enough to protect against basic attacks that alter data, it is not enough against more determined adversaries. If an attacker can accurately alter specific data, he may also recalculate and change its corresponding hash. In that case the data integrity may appear correct with a matching hash, even though it was altered by an unauthorized entity. To counter this, MAC functions also take a secret key as an input to generate their output. This way, only those who know the secret key could have generated a particular hash value for a particular message. The concept is shown in Figure 2.4. As long as the attacker does not know the secret key, generating a valid message and hash pair would be unfeasible. MACs as such can be used to ensure both the integrity and authenticity of data.

2.3. Application Risk Levels

As mentioned before, the application space of embedded and Internet of Things devices is rather broad. It ranges from small, battery powered sensor nodes to industrial controllers and from medical devices such as defibrillators and implants to military information systems at the front lines. Similarly, the risks they pose when compromised and subsequent security requirements vary wildly as well; from expendable to absolutely critical. Where one device could leak privacy sensitive information once under attack, another could disrupt an industrial process. While a hijacked weather measurement node could cause some mistakes in weather forecasts, the compromise of medical equipment could pose a danger

to human life.

As such, a distinction must be made between types of applications and their security demands. Though different fields will use similar types of devices (Sensor nodes are in use by greenhouses, weather agencies as well as the military for example) their realistic threats are vastly different. The rest of this section will distinguish three different groups based on these criteria. Low risk IoT covers most consumer electronics whose compromise will not lead to bodily harm. Next, medium risk will cover devices that do have the potential to cause harm or even pose a threat to human life when taken over. Finally, applications that could pose a danger to society and human life on a larger scale will fall in the high risk category.

2.3.1. Low Risk IoT

The low risk category is composed of applications and devices that will not have 'major' consequences or cause physical harm when compromised. This category includes most consumer electronics that are connected to the internet, as well as other applications that no vital systems depend on. Examples of such devices are printers [26], smart fridges, thermostats [9], video baby monitors [27], etcetera. Smart fridges can keep track of the products stored in it and alert the user when something is about to run out or reach its expiration date, an example of which is shown in Figure 2.5. A smart thermostat could learn the living rhythm of its owner and control the temperature in the house accordingly, and IP baby monitors or security cameras stream their video through the internet, allowing the owners to inspect their homes when away. Other applications, aside from consumer electronics, include sensor networks for weather agencies and GPS trackers that transmit their location and can be used to track the location of cars and other assets.

Figure 2.5: Interface of an LG smart refrigerator [28]

When devices from the group of consumer electronics are compromised, consequences may range from a loss of functionality to a breach of privacy. As an example, IP cameras may allow an attacker access to the video feed directly into the victim's home. Similarly, through internet connected thermostats an attacker might learn the living patterns of a particular target to find out when the house can best be robbed, as well as potentially raise the power bills by increasing the temperature when nobody is home [9]. Though not directly life-threatening, such a breach of confidentiality can lead to quite some discomfort and paranoia. In the case of corporate devices such as sensor networks and GPS trackers, there are some additional concerns. Aside from leaking sensitive data when a criminal or competing company gains access to the nodes, the attacker could also have the compromised nodes transmit manipulated 'sensor data' in order to disrupt operations at the victim. This would break the integrity, authenticity and availability of real data. If a GPS tracker can be made to transmit a wrong location or stop transmitting at all, the asset that was being tracked may now be stolen. Here, too, no lives are directly at stake, but the monetary losses may be substantial.

As the devices from the first group are commonly installed inside the homes of their users, the risk of

physical tampering is rather low unless the individual is personally targeted. Even then, the tampering must have been performed either before the device is delivered to the user, or the attacker will have to break into the victim's house to get access, both of which are not easily achieved. The main threats these devices face are coming from the network, for example from automated worms such as Mirai [4]. These generally exploit software vulnerabilities or default credentials left unchanged, to take over these devices and use them to form a botnet for DDoS attacks [29]. Due to the scale of the production as well as the profit margins of such devices, any changes to the supply chain is expensive. As such, devices may be shipped out with software that is months to years old, containing by now commonly known vulnerabilities to exploit [27]. In the corporate setting where corporate espionage and sabotage are a real concern, sensor networks face different threats. The software running on these devices is more tightly controlled by one entity, making software and default-credential based attacks less likely. However, because of their reliance on the network to transmit their data, operations may be disrupted by using jammers for example [30]. Furthermore, because of the decentralized nature of such nodes it would be easier to gain physical access to a device and mount hardware based attacks to bypass software security measures.

2.3.2. Medium Risk IoT

This category includes devices that could pose a real risk to human life when under attack, but not at a scale that would fit them under very high risk. The medical field is an example of such an application with pacemakers that can be read and updated wirelessly, defibrillators that report their health and battery status and perhaps the most visual application: Remote surgery. In the latter case a surgeon can control a set of robotic arms from anywhere in the world to perform surgery on a patient. Another application is in intelligent cars. Modern cars have all sorts of communication technology on board, ranging from over-the-air updates to media consumption for the passengers. Furthermore the central computer in a car, the ECU, is in charge of a lot of functionality. This includes engine management, ABS braking systems, powered steering and even the clutch and transmission. Some industrial applications controlling dangerous equipment could also fall in this category.

Figure 2.6: Result of a Jeep Cherokee being remotely taken over by researchers [5]

A person depending on medical devices could be at a very real risk when a pacemaker is remotely hacked, as shown possible in [31]. Imagining what happens when the connection is lost during a remote surgery because the hospital is hit by a DDoS attack is left as an exercise to the reader. Similarly, a car

can do a significant amount of damage when the driver loses control. Remotely executed attacks on the Jeep Cherokee pictured in Figure 2.6 have already been demonstrated, where researchers could disable the brakes, block the steering wheel and control the throttle, leaving the driver helpless [5] [20]. One well known attack on industrial processes is the American and Israeli Stuxnet worm. This worm was used in 2010 to disrupt the nuclear efforts of Iran where the virus caused Iranian nuclear centrifuges to destroy themselves without raising alarms [3]. In all of these cases, lives depend on the correct operation of equipment and tampering or attacks may well kill someone.

Somewhat depending on the application, these devices may be targeted specifically. Aside from personal attacks on someone by using their technology against them, a more determined attacker could extort companies by threatening to use found exploits. A company is likely to pay up to avoid the public outrage, if it became known that its cars can be made to kill its drivers. Such scenarios would make it attractive to criminals to actively look for any vulnerabilities in selected devices. The Stuxnet worm mentioned above was an example of a very specifically targeted attack, only affecting the particular controllers in use by the Iranian facility. To perform it, the hardware of and the software running on these controllers must have been extensively studied.

2.3.3. High Risk IoT

Finally, perhaps the most interesting, is the high-risk category. This covers devices that are likely to be targeted specifically by a powerful adversary and whose compromise can present a great danger to human life or society. Critical infrastructure is one example of such an application. This consists of power plants and the electricity grid, road networks and their control systems and bridges, water treatment facilities, trains, the internet itself etcetera. Similarly, industrial applications dealing with large amounts of volatile chemicals, as well as military applications fall under this high-risk category due to the inherent risks present.

The examples mentioned above are all important strategic targets. If a city were to be left without power or clean water for extended periods of time, chaos would ensue. Attacks on such infrastructures have already been recorded outside of laboratory settings. In 2015 for example, the Ukrainian power grid was attacked and partially taken down for several hours, leaving hundreds of thousands of people without power [32]. There was a military incident in 2011, when Iranian forces took control over an American drone, landed it and reverse-engineered its technology [33]. A picture of the captured drone is shown in Figure 2.7. Stuxnet mentioned above showed that industrial controllers can be targeted. This means the same holds true for those used in industrial processed with dangerous chemicals. Forcing those to vent toxic materials into their surrounding could cause an ecological disaster.

Figure 2.7: Picture of the US drone that was captured by Iran [33]

Because of their strategic value, these are attractive targets to both terrorists and rivalling countries waging cyber wars. Especially the latter will have a great deal of resources and manpower available to research and abuse potential weaknesses in an enemy's systems. They would have high-end equipment to analyze hardware, high-performance computer systems to run tests on software and whole teams within secret service departments working on analyzing networks. As such, devices in critical

applications require the highest possible level of security.

3

Overview of Relevant Attack Types

Now that the properties, applications and risks associated with the Internet of Things have been covered, this chapter will discuss several attacks that threaten such devices. These attacks are grouped into three categories, based on their main attack vector. In each category, multiple attack types relevant to the IoT are explained, some examples are provided and, if any, potential countermeasures are mentioned. Section 3.1 first covers attacks from the network level. Next, Section 3.2 presents attacks targeting the software. Finally, Section 3.3 goes over four attacks that are performed directly on the hardware of a device through physical access.

3.1. Network Attacks

This section will cover four network based attacks relevant to the Internet of Things. Such attacks can be performed remotely, without needing access to the target device. As they are typically mass produced, many identical devices are put at risk when an attack is found successful against one of them. Furthermore, targeting the network itself can disrupt the operation of all connected IoT devices in various ways. The methods with which a device is attacked depends on the specific goals of the attacker, be it stealing data, sabotaging a rival company or setting up a botnet.

First, Section 3.1.1 will explain port scanning and brute-forcing credentials, a common vulnerability that is often exploited by later attacks. Next, Section 3.1.2 will cover some ways an attacker can accomplish a Denial of Service (DoS) by limiting the availability of devices connected to a network. Section 3.1.3 shows how battery powered nodes can be targeted specifically to drain their batteries and shorten their operational lifetime. Finally, Section 3.1.4 will show some examples of what a Man in the Middle (MitM) attacker could do once he has gained access to a node within a network.

3.1.1. Scanning/Brute-Force

By far the easiest way to take control over a device, is by having valid credentials and simply logging in. To allow users and/or owners of an IoT device to control its actions, there must be some way to access it over the network. There are several ways to do this, but for their security they all rely on one fundamental criteria: The correct combination of username and password must only be known to legitimate users. Once this criteria no longer holds and an attacker gains knowledge of these credentials, all sense of security is lost. He can login as a valid user and take full control without needing to resort to any more complicated exploits and attacks.

There are plenty of tools available to attackers to figure out what kind of login service is available on a device. One such example is the open source $nmap$ application [34]. Nmap is a network scanner that is meant to find hosts connected to a network, as well as finding out what services are running on said hosts by scanning it for any open ports. An example output of nmap is shown in Figure 3.1. Though it is developed for legitimate purposes, an attacker could use it to find targets on a network as well. Once a target and its interfaces are identified, the next step is to get credentials.

Though the aforementioned criteria where only legitimate users know the password should be a good protection in theory, in practice it often does not hold. Many consumer IoT devices are shipped with default account names and passwords which cannot be changed or disabled by the end-users.

```
[admin@Ter ~]$ nmap -A -T4 scanme.nmap.org
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-07 14:37 CEST
Warning: 45.33.32.156 giving up on port because retransmission cap hit (6).
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.16s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 994 closed ports
PORT      STATE    SERVICE      VERSION
22/tcp    open     ssh          OpenSSH 6.6.1p1 Ubuntu 2ubuntu2.13 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   1024 ac:00:a0:1a:82:ff:cc:55:99:dc:67:2b:34:97:6b:75 (DSA)
|   2048 20:3d:2d:44:62:2a:b0:5a:9d:b5:b3:05:14:c2:a6:b2 (RSA)
|   256 96:02:bb:5e:57:54:1c:4e:45:2f:56:4c:4a:24:b2:57 (ECDSA)
|_  256 33:fa:91:0f:e0:e1:7b:1f:6d:05:a2:b0:f1:54:41:56 (ED25519)
80/tcp    open     http         Apache httpd 2.4.7 ((Ubuntu))
|_http-server-header: Apache/2.4.7 (Ubuntu)
|_http-title: Go ahead and ScanMe!
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
9929/tcp  open     nping-echo   Nping echo
31337/tcp open     tcpwrapped
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 37.01 seconds
```

Figure 3.1: Example output of an nmap run, note the ssh service on port 22

Furthermore, even if a password is unique to a device, the algorithm with which it was generated may be weak, allowing an attacker to guess the password with little effort [27]. As such, brute-forcing passwords is a realistic strategy.

Examples

Researchers at Rapid7 [27] found several vulnerabilities in commercial video baby monitors that are trivial to exploit. Many of them were caused by known default login credentials to the devices, enabling attackers to take control over the device through ssh or telnet. Common username and password combinations include $admin - admin$ and $user - 12345$, making it easy to brute-force.

In 2015, researchers demonstrated that they could take full control over a Jeep Cherokee car over the internet, through an insecure media entertainment system [5]. This system generates a 'unique' password based on the time, down to the second, it was first turned on. Though this would seemingly leave many options, they estimate that it would take less than half an hour to brute-force a year worth of passwords. Furthermore, it appeared that the system time is set based on received GPS time. If this time could not be determined before the password is first generated, i.e. within a minute or so as was the case during their experiment, it would default to a hardcoded time, meaning brute forcing is trivial.

In 2016, the Mirai worm rapidly managed to infect hundreds of thousands IoT devices which it used to perform some of the largest DDoS attacks on record [4]. It would scan random IP addresses for open Telnet TCP ports 23 and 2323 and, when found, would enter a brute-force phase. During this phase, it attempted to login with 10 common default username and password combinations picked from a list of 62 pairs. After Mirai's source code was published online, several derivatives using the same methods have been active as well.

Countermeasures

To protect against scanning and brute-force attacks, there are some countermeasures.

Firstly, an attacker can not attack what does not exist. Closing ports and disabling services that are not absolutely required for operations and maintenance, limits the possible attack surface of a device. Unencrypted protocols such as Telnet should be avoided for administrative access and ssh should be used instead.

Second, a limit could be put on wrong login attempts after which there is an increasing delay before it can be tried again. This would prevent an attacker from rapidly trying many different credentials. One downside of this technique is that it might lock legitimate users out of the device while an attack is happening.

Finally, strong and unique login credentials would counter brute-force attacks. These would include random usernames and passwords, not generated through some easy to guess method [27]. Better yet, a public/private keypair could be used to log in to ssh, where each device generates its own keys upon first activation.

3.1.2. Denial of Service

The classic image of a Denial of Service (DoS) attack is to flood some high level website with millions of requests. This then overwhelms the network and servers and consequently limits or completely blocks any connections from legitimate users. In the context of the Internet of Things, the techniques behind a DoS attack can be quite different, though the result is the same: The availability of a service is drastically reduced. When a network of sensor nodes in a greenhouse is targeted by an DoS attack for example, climate control systems relying on the data of said sensors may fail and crop yield may be affected.

The reasons for an attacker to perform a DoS attack may vary. A company could try to sabotage the profits of a rival, it may be part of an extortion scheme where the victim must pay the attacker to regain access to their assets [35], or perhaps it is part of a political statement as were some DoS attacks on government websites by Anonymous and other groups [36]. Either way, this type of attack poses a real threat to IoT networks.

Examples

Often IoT devices are the edge components of a larger system, with a central server with which they communicate. As such, targeting this central server can bring down the entire network. This may be done through a Distributed DoS (DDoS) attack, utilizing many devices to flood the server with traffic and bring it down similarly to attacks on websites mentioned above [37].

IoT devices in the field that communicate and connect to the internet wirelessly are at risk of jamming. Any radio communications could be disrupted by another source, transmitting at the same frequency at the same time. Jammers may be either proactive; transmitting at random frequencies at random times, or reactive; listening to the wireless channel and begin transmitting at a given frequency once a packet transmission is detected [30].

Countermeasures

Countermeasures against DDoS attacks targeted at websites are a well studied topic. Some solutions range from dynamic load balancing with extra servers, to requiring a valid response from the attacker to a computational challenge before the server looks at their messages. These may however be less suitable to IoT applications. Researchers in [37] show how to use a $honeypot$ to direct any distributed DoS attacks away from the actual base station.

The amount of research performed into countering jammers, too, is extensive. Researchers in [30] let other nodes in the network transmit decoy messages to which the reactive jammer may respond, allowing other nodes to transmit their real messages in another channel. Namvar et al. propose an evolutionary algorithm to maximise successful message transmissions by spreading its power over the available channels [38].

3.1.3. Denial of Sleep

Denial of sleep is an attack specifically targeted at autonomous, battery powered sensor nodes. It is in a way a type of Denial of Service attack in the sense that it reduces availability of a system. Its effects however are longer lasting and once the damage is done, it requires a costly and manual labour to become operational again. As such they deserve to be covered in a separate section.

As they run from batteries, low power sensor nodes spend most of their time in a nearly off state to reduce power consumption to a minimum. Only at specific intervals or when receiving a specific command do they wake up, perform their measurements, and transmit their data before going back to sleep. This allows them to be placed in hard-to-reach places without nearby power outlets, while still remaining operational for several years [19]. On the other hand, this does make them vulnerable to attacks specifically aimed at preventing them from entering sleep mode and as such rapidly draining their power supply. When their battery has been drained, they will remain inoperable until someone is sent by to replace it.

Examples

The goal of a sensor node is to occasionally turn on its radio, transmit some measurement data and power back down to sleep mode. After sending its data, it waits for a reply from the base station to verify that the data was correctly received. If this reply is blocked, for example by jamming, the node will re-transmit its data. Researchers in [39] provide several methods to do just that. By keeping the node and its radio awake, precious power is consumed, draining its batteries.

Countermeasures

As the attack in practice is a specific type of jamming, the same countermeasures mentioned above in Section 3.1.2 are valid. Alternatively, if their environment allows it, devices could be supplied with a method to generate their own power, for example with a small solar panel.

3.1.4. Man in the Middle

A Man in the Middle attack (MitM) is a rather broad term in the context of IoT devices. In short, a MitM attack is when an adversary can intercept data transmitted between two devices unbeknownst to them, at which point it can record or manipulate it. The goal here could be to steal sensitive information, to tamper with measurements or simply to prevent the information from reaching its target.

Examples

One relevant type of MitM attack on IoT devices are so called Sybil nodes. During a Sybil attack, an attacker impersonates legitimate devices in the network, in order to sabotage the effectiveness of the system. Compromised or custom nodes are inserted into the network, and communicate with legitimate nodes as if they are part of the network. Zhang et al. provide an overview of different types of Sybil attacks [40], and an overview is given in Figure 3.2. The main goal of these attacks is sabotage of processes.

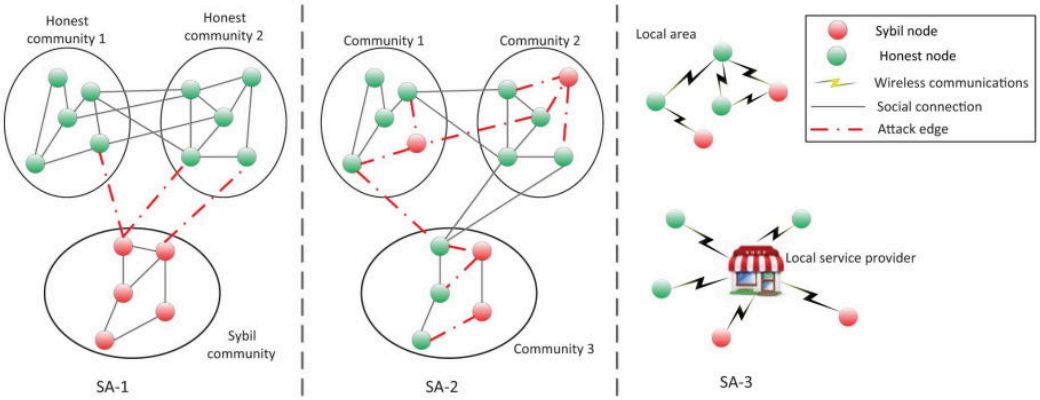


Figure 3.2: Schematic overview of types of Sybil attacks [40]

Countermeasures

Several countermeasures against Sybil attacks exist. SCUBA, for example, lets the base station verify the integrity of its nodes to detect if one has been compromised [14]. Similarly, digital signatures can be used during communications, to ensure that data originates from a legitimate source and has not been tampered with. or additional and more advanced solutions, the reader is referred to the following paper, covering Sybil detection methods and defences [40].

3.2. Software Attacks

This section will cover three typical software-based attacks. Rapid7 found that IoT devices are often shipped and sold with outdated software [27]. As they are mass-produced, this leads to many potential targets that run software with known vulnerabilities. Once an attacker manages to run some malicious software on a target device, there are various ways to take control over it and the consequences could be severe. As with the network based attacks mentioned above, we will start with a particular vulnerability in applications that is often abused as part of an attack, the buffer overflow, in Section 3.2.1. Next, perhaps the most famous types of software attack, worms and viruses, will be discussed in Section 3.2.2. Finally, some techniques regarding firmware injection will be mentioned in Section 3.2.3. For more types and insight in software based vulnerabilities and attacks, the interested reader is referred to one of many taxonomy papers [41] [42] [43].

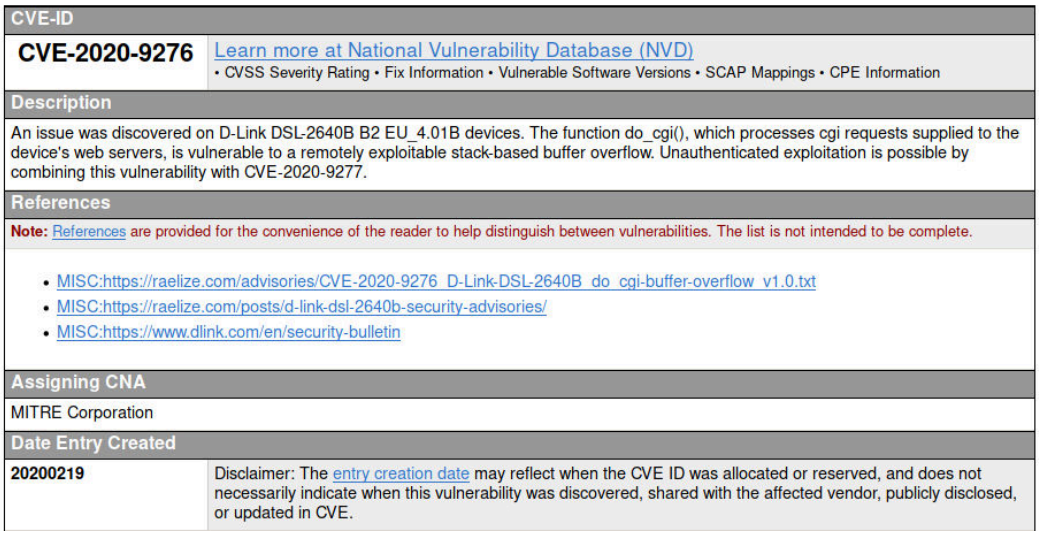
3.2.1. Buffer Overflows

Software attacks are generally possible because of some vulnerability in the implementation of the programs running on a device. These vulnerabilities can allow an attacker to crash a process, run malicious code or bypass security measures. A large and constantly growing list of Common Vulnerabilities and Exposures (CVE) in various applications is kept by Mitra Corporation [44]. In 2015, Papp et al. went through this list and created a filter to focus on entries related to embedded systems [41]. This way they proposed another taxonomy of attacks to such systems based, amongst others, on the type of vulnerabilities exploited to work. Similarly, researchers in [45] identify seven common causes of security problems that arise from coding mistakes as well as ways to prevent them. These bad practices range from failing to validate inputs to badly handling errors, by having them expose too much information to possible attackers. Finally, both [46] and [43] mention buffer overflows as common methods to make programs perform unwanted behaviour in the context of embedded devices, supported with several CVE entries.

Buffer overflows are a particular type of vulnerability where failing to check and sanitize inputs can result in unintended parts of memory being overwritten, by providing a larger input than expected. Doing so on purpose could allow an attacker to change the return address of a function, or even to inject executable code directly [47]. Exploiting a buffer overflow can have effects ranging from crashing a process, all the way to privilege escalation and remote code execution [46]. A lot of these vulnerabilities can be avoided by adhering to good programming rules [45].

Examples

At the time of writing, there are nearly eleven-thousand CVE entries related to buffer overflows mentioned in Mitre's database [44]. An example of one such entry can be seen in Figure 3.3. The reader is referred to this list for a complete overview of known attacks, though some will be mentioned here as well.



CVE-ID

CVE-2020-9276 Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

An issue was discovered on D-Link DSL-2640B B2 EU_4.01B devices. The function do_cgi(), which processes cgi requests supplied to the device's web servers, is vulnerable to a remotely exploitable stack-based buffer overflow. Unauthenticated exploitation is possible by combining this vulnerability with CVE-2020-9277.

References

Note: References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MISC:https://raelize.com/advisories/CVE-2020-9276_D-Link-DSL-2640B_do_cgi-buffer-overflow_v1.0.txt
- MISC:https://raelize.com/posts/d-link-dsl-2640b-security-advisories/
- MISC:https://www.dlink.com/en/security-bulletin

Assigning CNA

MITRE Corporation

Date Entry Created

20200219 Disclaimer: The entry creation date may reflect when the CVE ID was allocated or reserved, and does not necessarily indicate when this vulnerability was discovered, shared with the affected vendor, publicly disclosed, or updated in CVE.

Figure 3.3: Example CVE entry of a buffer overflow (CVE-2020-9276)

One example where a buffer overflow vulnerability can have severe consequences was presented by Hanna et al. [6]. They discovered multiple vulnerabilities and attacks against a defibrillator, amongst which a buffer overflow in the firmware update software. Exploiting it allowed arbitrary code execution, leading to dangerous situations for patients.

In October 2016 the source code of the Mirai worm, infecting many IoT devices and responsible for some of the largest DDoS attacks so far recorded [4], was published online. Interestingly it was found that the worm itself suffered from multiple vulnerabilities, including a buffer overflow. Exploiting it would crash the process, keeping it from attacking further [46].

Countermeasures

As buffer overflows generally originate from implementation errors, this is also where best to fix them. By performing proper input validation, any inputs that are too large or otherwise of the wrong type may

be filtered out. Good coding practices without relying on the assumption that inputs will always be within an expected range, can prevent buffer overflow vulnerabilities from occurring [45].

Another method are to use so called stack canaries [47]. They can be implemented through a basic compiler patch and allow for buffer overflows to be detected and even countered, without requiring to change any source code. One way implement them is by putting a certain value, the 'canary', before the return address in memory. Then before the function returns, the value of the canary is checked to see if it has been overwritten by a buffer overflow. If it has, an alert can be triggered.

3.2.2. Viruses and Worms

Viruses are perhaps the most well known software based threat to computer systems, mostly because they affect the general users. They are self-replicating programs that spread by attaching itself to another file or program. Whenever this file is opened or the program is ran, so is the virus, allowing it to spread and infect more files. When an infected file is stored on a portable storage medium such as flash drives, the virus may spread to other devices when the drive is connected to them. In the early days of personal computers, infected bootsectors of floppy diskettes were a common way for viruses to spread [23].

Worms are a threat similar to viruses, the main difference being their method of spreading and infecting other devices. Where viruses use files and storage media, worms spread themselves over a network [42]. They could be automated bots looking to login and install itself on unprotected devices, mass-email themselves to potential victims hoping they will download the attachments or otherwise spread themselves over the internet. The fact that they spread through a network could also classify them as a network-based attack, but due to their similarities to viruses as well as the fact that they are based around malicious software, they are covered together.

Once a virus or worm has infected a device, it can deploy its payload with various effects: It may delete or alter files, leak sensitive information or even demand a ransom for encrypted harddrives. Because of the nature of IoT devices; autonomous, lightweight and with an internet connection, worms in particular are a real threat. Some examples of and countermeasures to some known attacks are provided next.

Examples

In 2010, the Stuxnet worm marked a new era in digital warfare when it infected and destroyed several of Iran's nuclear centrifuges. Though neither admit being involved, it is regarded as being developed by the United States and Israel to delay Iran's nuclear program. Initially infecting its first targets through USB drives similar to a virus, it would then spread over local networks and infect all windows PC's it could reach by using four different zero-day exploits in Windows. It would then look for particular controller with specific configurations, ensure it was a valid target, and then install its payload. This way, though infecting over 100.000 machines worldwide, only the controllers in Iran are known to have been affected by the specific payload. Once installed, the payload would let the centrifuges destroy themselves by spinning at excessive speeds, while replaying older normal data to the operators [3].

The Mirai worm as mentioned before rapidly infected many IoT devices to perform large-scale DDoS attacks in 2016. It spread by scanning random IP addresses for open Telnet ports, and would then attempt to login with a set of default username and password pairs. It specifically targeted unprotected consumer electronics and IoT devices such as a large number of IP cameras [29]. Once a vulnerable device was found, it would report back to a control server and then download and install some device specific malware on it. Afterwards it would take steps to hide itself, disable all other processes using Telnet or ssh to block other worms from infecting the device, and then wait for commands while scanning the internet for new targets [4] [46].

In 2017, the WannaCry worm infected over 200.000 computers worldwide. It spread using the EternalBlue exploit affecting Windows computers, which was developed by the NSA and became part of a leak in 2016. Once it had infected a device, it would encrypt data on its hard drives and then scan the network for new potential targets. It would then display a message demanding the victim to pay a sum of bitcoins to some given wallets in order to receive a decryption key. This way it crippled the British National Health Service and caused several factories to shut down for some time [48].

Countermeasures

There are several methods to prevent viruses and worms from spreading, a few of which will be briefly mentioned next:

Figure 3.4: Screenshot of ransom message from the WannaCry worm [48]

As with other software-based attacks, the most straightforward method is to regularly apply software updates. These updates may fix potential security issues as they are found, preventing any malicious software from exploiting them. Though this may not be enough if the worm is using zero-day exploits as was the case for Stuxnet [3], Microsoft did release a patch against the vulnerability used by WannaCry two months prior to its attack [48] .

Worms that rely on unsecured services such as Mirai can be simply stopped by changing any default login credentials [4].

Antivirus software can detect the operations of worms and viruses, and alert the user or stop the process. Here, too, regular updates must be applied to keep the detection database up to date with the latest and newest threats.

3.2.3. Firmware Injection

Vulnerabilities in software may allow an attacker to run arbitrary code. When this gives him access to the device's update mechanisms, firmware injections are possible. Here, custom firmware can be provided and installed as an 'update', after which the attacker has full control over the device. If the update mechanisms themselves have weak security, e.g. if updates are fetched in plaintext from a ftp server, legitimate updates can intercepted, inspected for weaknesses and modified.

Examples

Cui et al. showed a clear example of insecure software allowing access to the update mechanisms of an HP printer [26]. Files to be printed could be provided to the printer either over the network or directly through USB. The printing subsystem, however, was coupled with the firmware update mechanism. The researchers were able to provide a modified file to the printer as a print job, which then detected it being a firmware update and went on to install it without need for any authentication.

In this case, they extracted the installed firmware through a hardware attack, by removing the flash chip from the device and dumping its contents as shown in Figure 3.5. The firmware could then be studied for vulnerabilities. They estimate tens of millions of devices to be vulnerable to such attacks.

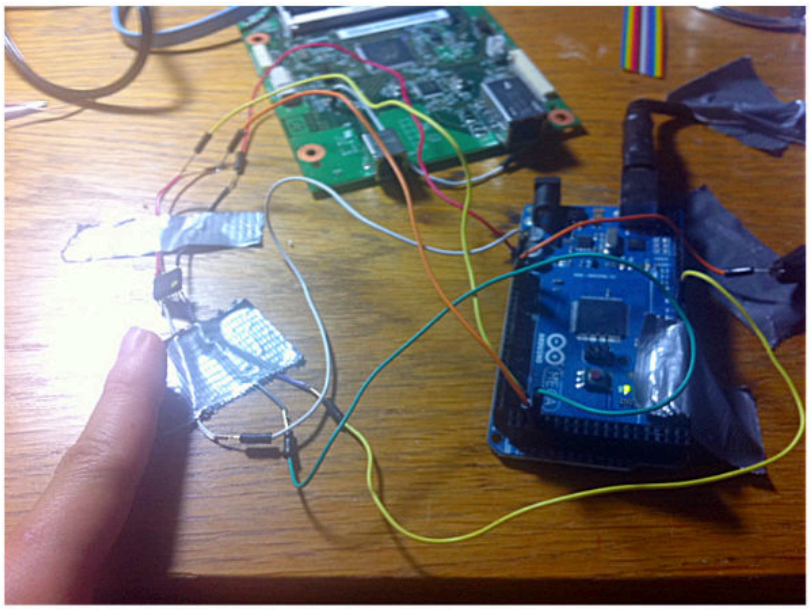


Figure 3.5: Physical removal of flash chip from a device to dump and study its firmware [26]

Countermeasures
Arm TrustZone [12] could be used to separate the different subsystems from security critical components such as the update mechanisms. This way, only updates that are signed by the manufacturer can be installed. Similarly, encrypting the firmware updates would make it more difficult to find vulnerabilities if the download was intercepted, and to generate valid modified updates. Note however, that being signed and encrypted does not necessarily imply that the official update will not have vulnerabilities to exploit anyway.

3.3. Hardware Attacks

Hardware based attacks form a completely separate group from the previous two. Where network and software based attacks overlap and can generally be performed remotely, hardware based attacks per definition require the attacker to have access to the device. Although this puts a limitation on what types and how many devices can be attacked at once, it does give the attacker a strong advantage. Devices can be analysed in detail for any open vulnerabilities without having to rely on known bugs in its software.

This section will cover several attack types and explain how they work, give some examples and discuss what is already done to protect against them. Probing is mentioned first in Section 3.3.1, where an attacker manipulates any available interfaces of a device. Following this, fault injection is discussed in detail in Section 3.3.2. Side-channel analysis is used to passively learn more about what happens inside a device by closely observing its surroundings, and will be covered next in Section 3.3.3. Finally hardware trojans, which are a completely different type of attack, will be mentioned in Section 3.3.4.

3.3.1. Probing

Embedded devices generally have some unused ports on their boards that were used for debugging by the designers. When these ports are left active and available on production versions, these may be accessed and abused by potential attackers. Especially since they give low-level access to the device, often bypassing software security measures, they are an attractive target. Some examples and countermeasures against their miss-use are discussed next.

Examples
Most if not all interfaces of a device can be hooked up to an oscilloscope or logic analyser to observe the data flowing through them. If done on the traces to and from external memory, for example, the contents can be recorded without needing to read directly from the memory chip.

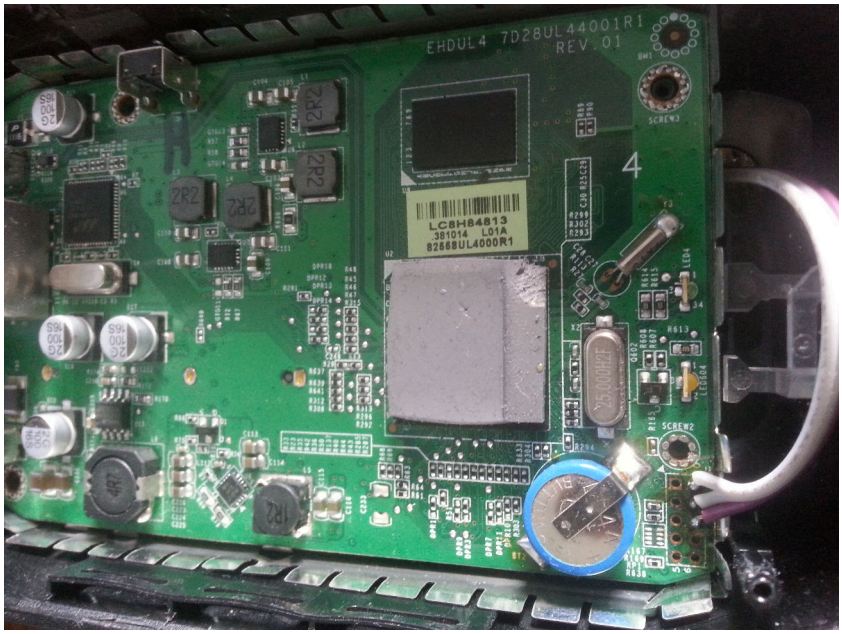


Figure 3.6: Internals of a Seagate GoFlex NAS, note the exposed UART on the bottom right

Some interfaces are simply left-overs from development. UART to debug, JTAGs to program, SPI for external flash that was scrapped later in the design, all may still be available by probing the right pins. An example of this can be seen in Figure 3.6. This picture shows the main circuitboard of a Seagate NAS, with exposed UART headers in the bottom right. Connecting wires to it as in the picture, instantly grants access a root shell on the device without any passwords. Rapid7 reported similar situations for most commercial IoT devices such as IP cameras [27]. With this level of access, all other security measures can be considered defeated.

Microcontrollers are typically able to boot from multiple sources, selected by providing specific voltages to external boot pins during startup. These options can include booting from internal flash, from an external SPI connection, through UARTs, USB etcetera. One example of these boot pins being used to load custom firmware, is the research by Hernandez et al. on the Nest smart thermostat [9]. By forcing a boot from USB, they were able to circumvent any firmware verification of the device.

Figure 3.5 shows how the researchers of the firmware-injection attack on printers, mentioned above in Section 3.2.3, removed the printer's flash chip. This chip was then hooked up to an arduino in order to extract its contents and study the firmware for vulnerabilities.

Countermeasures

Though the obvious solution of not routing pins and traces on a board for functionality that won't be used in the final design could solve the issue, it is not ideal. Doing so would make development more difficult and require additional production runs. One option is to include security fuses in the chips that can be burned through upon programming, similar to single-write read-only memory. After development, these fuses can be burned through to physically and permanently disable the interface.

Some interfaces however cannot be disabled because they are needed for operation. These include the pins and traces to external memory modules, that contain the device's firmware and RAM. In order to protect these modules and their contents, additional measures are needed.

3.3.2. Fault Injection

Fault injections are a type of hardware attack where the attacker deliberately induces a fault in a system, in order to observe the response and gain access to secret information [49]. The goal of the attack could for example be to change bits controlling security checks, privilege levels or bootloader addresses. By doing so the attacker may be able to skip steps during an encryption algorithm and leak information about the secret key. Alternatively it could allow the attacker to circumvent security measures against booting from unknown sources, and insert his own modified programs. Many ways of performing fault

injection attacks exist in the literature with varying levels of accuracy, complexity and cost. Some will require expensive lab equipment, whereas others do not need much more than a light bulb. This section covers some known techniques as well as some countermeasures. For a more extended classification of attacks and countermeasures, the reader is referred to Bar-El et al. [50].

Examples

One well known method of fault injection is by manipulating the power supply of a device. When the supply voltage is lowered, the setup time of logic gates increases. With the clock frequency being kept the same this will eventually cause the slower logic paths to fail and a fault to occur. Barenghi et al. used this technique to break the AES cipher running on a common $32bit$ ARM processor [7] without any special equipment. This type of attack can be extended with correctly timed power spikes as shown in [51]. There, the authors created over-voltage spikes while writing data to an RFID tag, causing it to store corrupted data to its memory. Similarly Timmers et al. were able to load values of their choosing into the program counter register of a commercial ARM SoC by glitching the power line during a load instruction [8]. This allowed them to bypass any secure boot mechanisms built into the system. Though more advanced than the first example, this type of attack can still be performed with inexpensive hardware.

Manipulating the clock of a device is a related technique. By increasing the frequency or inserting clock glitches, an effect similar to that of a reduced input voltage can be achieved: Slow logic paths can be made to fail by not allowing enough time for their correct setup. Balasch et al. demonstrated an example of this in [52], where glitching the clock input of an $8bit$ AVR micro-controller enabled them to replace and skip instructions. It is clear that this type of attack is only possible on devices with an accessible external clock signal, which limits its applicability on modern embedded systems with internal clock generation. Furthermore, the equipment that is used to generate the clock glitches needs to run at a (much) higher frequency than the targeted device.

Another method to generate faults is by overheating the target device, causing the internal components to become unreliable. The authors of [53] broke the protection mechanisms of Java Virtual Machines by inducing errors in a commercial desktop PC's DRAM. After considering waiting for cosmic rays to cause memory errors and suggesting to buy a small oil company to get access to its restricted Americium-Beryllium source, they decided against this on the grounds of not having the time to learn the oil drilling trade. Instead they showed that the same was possible with nothing more than a thermometer and an incandescent light bulb aimed at the DRAM chips. A picture of their experimental setup is shown in Figure 3.7.

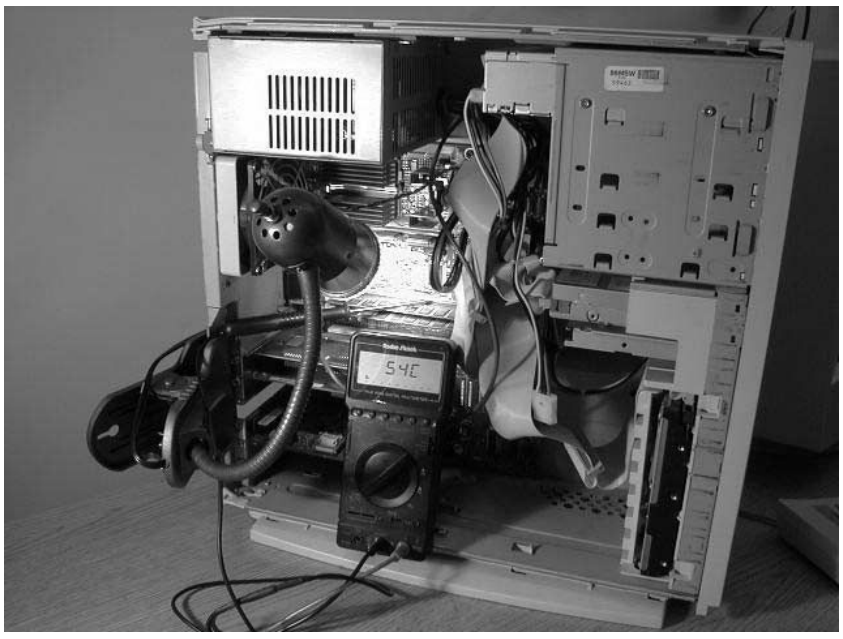


Figure 3.7: Experimental fault injection setup using the heat from a common light bulb [53]

Further, more advanced methods include inducing faults through electromagnetic radiation. In aerospace applications electronics are continuously bombarded by cosmic rays and electromagnetic waves. These rays and waves have enough energy to trigger transistors in integrated circuits, causing bit flips and subsequent transient errors [54]. As transistors have become smaller and smaller over the years, these effects now also play a role at ground-based electronics. In [55], Moro et al. show in detail how to affect a micro controller by purposefully exposing it to electromagnetic waves generated by high-voltage pulses through a coil. Similarly, in [56] the authors use electromagnetic pulses to set and reset D-flip-flops inside an FPGA. They then propose a sampling-fault model to explain the resulting experimental observations. This method of fault injection does not require the attacker to alter anything about the targeted device, thus leaving no traces. Pulses can be sent into the device by attaching probes to its housing. Figure 3.8 shows their advanced setup. It includes 3-axes vision and positioning systems, as well as an oscilloscope and pulse generator.

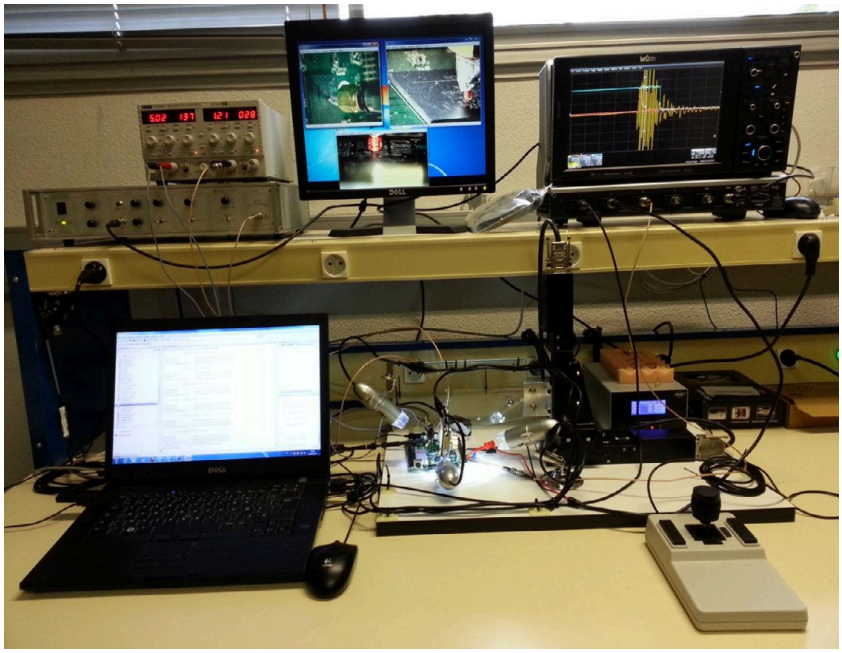


Figure 3.8: Experimental fault injection setup using electromagnetic pulses directed at a device [56]

Optical fault injection is related to the aforementioned EM method in that it uses waves to flip bits in integrated circuits. This time however the waves are generated by UV light, lasers or a camera flash unit as used in [57]. There, Skorobogatov et al. decapsulate the target chip to gain access to the chip surface itself, which they call a $semi - invasive$ attack as the passivisation layer of the chip is left undisturbed. They were able to change any individual bit in an SRAM cell to their choosing by focusing a flash of light at the correct area. The limiting factor in their attempts was the accuracy achievable with their camera flash as it does not produce light of only one wavelength, thus making it difficult to aim. This limitation was solved by using a commercial laser pointer instead.

The last group of hardware fault injection attacks that will be covered here are paradoxically software based. Although the attack itself is not performed with tools directly influencing the hardware of the targeted device, the faults themselves are very much hardware based. One well known example of such an attack is $RowHammer$. Its operation is based on the increasing memory cell density allowing more and more data to be stored in a single DRAM chip. As the DRAM cells get smaller and closer together, parasitic coupling effects between cells become relevant. This leads to a vulnerability where memory rows are influenced by what happens to adjacent rows. Kim et al. showed in 2014 that data in one DRAM row can be altered and corrupted by repeatedly accessing a neighbouring row, and that 110 out of 129 of the tested memory modules is affected by this vulnerability [58]. This poses a security problem where one malicious application can influence the data of any other application.

Countermeasures
As fault injection attacks can take a wide variety of forms, so do their countermeasures. Some of these methods will be briefly covered next.

Supply voltage detectors can be used in the chip to detect variations in the supply voltage and raise an error when the input voltage exceeds certain thresholds [50]. A frequency detector can be used the same way to detect any variations in the clock signal [50]. This way an attack can be detected as it occurs, allowing the device to reset or shut down. Similarly, decoupling capacitors may be used as a buffer against sudden input spikes by keeping the supply voltage stable.

Countermeasures against the more advanced EM and light based fault injection techniques may include using dual-rail logic and tamper sensors inside the chip. When a fault is generated in one of the lines, it can be readily detected by comparing it with the other line [57]. A light detector could be one type of tamper detector which is triggered when the chip is decapsulated for the attack [50]. Similarly, some type of wire mesh in the top layers of the silicon could be used to detect a change in capacitance once the packaging is removed.

Another possible countermeasure to these attacks would be either hardware or software redundancy [50]. Redundancy here means that the same operation could be performed multiple times, either in parallel in hardware or sequentially in software. Under normal circumstances the results of both executions should be identical. If they somehow differ, something must have happened and an alarm can be raised. This redundancy does come at the cost of either an increase in hardware size and power consumption, or an increase in execution time.

Finally in the case of the aforementioned $rowhammer$ attack, the authors proposed several possible solutions [58]. These range from the manufacturers making 'better' chips to frequently refreshing all rows in memory to prevent them from influencing each other. The authors' preferred solution is referred to as PARA (probabilistic Adjacent Row Activation) where each time one row is accessed, there is a probability that one of its adjacent rows will be refreshed. This would prevent a $rowhammer$ attack as a row becomes likely to be refreshed when its neighbour is frequently accessed, without the overhead costs of constantly refreshing each row.

3.3.3. Side Channel Analysis

Though the techniques mentioned in the previous section seek to expose hidden internal behaviour of a device by actively generating faults, Side-Channel Analysis (SCA) is all about gaining information about the internal operations through passive means. By carefully observing the targeted device and its surroundings all sorts of information may be leaked about internal states, cryptographic keys or other secret data. Where fault injection induces variations in voltage, temperature or electromagnetic fields, SCA instead closely monitors these points for variations caused by the device itself.

Examples
In 1998, Kocher et al. were the first to perform power-based side-channel attacks. They recorded the power consumption of a device performing DES encryption [59], of which an image is shown in Figure 3.9. Based on the data and cryptographic key, conditional statements were sometimes executed and sometimes skipped, which is visible in the power consumption. This way, they were able to extract secret information, and to break the security of the device.

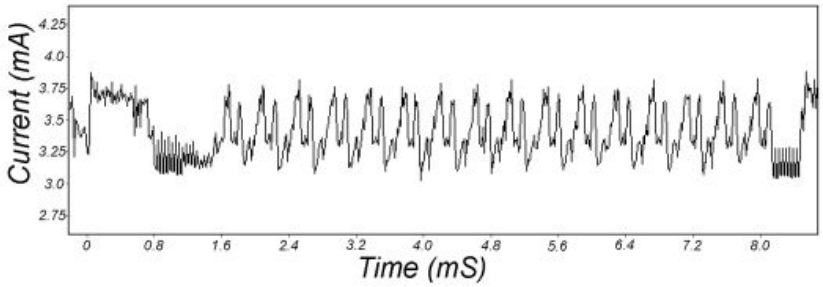


Figure 3.9: Variations in power consumption of a smart card during a DES operation [59]

Since then, side-channel analysis methods were developed based for example on differences in execution time [60], variations in the electromagnetic field around a device [61] and many others.

Countermeasures
Countering side-channel analysis is a rather broad and implementation specific field. One basic example to counter power and timing based analysis is performing redundant computations. As mentioned above, these differences are caused by software branches or hardware being used or not, depending on the secret information. By always executing all branches but only using the resulting data if needed, all runs will have the same duration and power consumption no matter the data, at the cost of additional energy being used [62]. Electromagnetic radiation could be made harder to detect, by shielding cryptographic hardware by ground planes inside the chip. For a more detailed overview of possible countermeasures, the reader is referred to one of many papers on the subject [62].

3.3.4. Hardware Trojans
Hardware Trojans are changes or additions made to the design of a device unbeknownst to the original designers, in order to cause malfunctions, leak sensitive data or otherwise maliciously influence the performance once the device is deployed [63]. Whereas the previously mentioned hardware attacks are generally performed once a device is deployed in the field, hardware trojans are installed during earlier phases in development. Due to the length of the IC supply chain, there are many places a hardware trojan could be included: First, someone in the original design team could have bad intentions or be otherwise 'convinced' to include one. Alternatively a third-party IP core could have something hidden inside without the designers being able to check. The proprietary and closed source design tools themselves might be compromised, or maybe the foundry actually making the chip could not be trusted [64] [65].

Wang et al. proposed a classification scheme for hardware trojans based on six attributes: Four different physical characteristics of the hardware trojan -its type, size, distribution and structure-, its method of activation and its effects once activated [66]. The type here specifies whether the trojan is realized by adding or removing transistors from the design, or by purposefully modifying existing wires and transistors in order to reduce functionality or performance. Its size is straightforward and covers the amount of components that have been added or altered in the design. Distribution describes where the hardware of the trojan is located in the chip; either topologically close together or spread out loosely throughout the silicon. The structure mentions whether or not the layout of the target chip needs to be regenerated in order to include the trojan, which would make it more difficult to hide, or if it can be hidden in some empty spot. The activation method is a broad category and can range from always on to condition based, and from internally to externally activated by several stimuli. Once activated, a trojan will perform its intended action. These are classified into modifying functionality, specification or leaking information to the attacker.

Examples
Although to the best of my knowledge no real-world examples of hardware trojans embedded in IC's have ever been found and published about, a significant amount of literature has been written on the subject. Most of this literature deals with either possible ways of constructing trojans, or methods of detecting them in a final product. Figure 3.10, for example, shows various ways a hardware trojan could be activated.

Countermeasures
So far no method has been found in the literature that can fully prevent the installation of a hardware trojan when parts of the supply chain cannot be trusted. The ability to detect the presence of a trojan however has been studied in detail. Tehranipoor et al. for example categorize known ways of detecting a trojan in a device [67]. The authors split detection methods into side channel analysis and attempting to activate any trojans that might be present. With the first method, side channel analysis is performed on a number of chips registering characteristics such as timing and power draw. These chips are then destructively reverse engineered to ensure that no trojan is present. Next, the side channel characteristics of the chips-under-test are compared to those of the known 'clean' chips. Any deviations from these 'clean' characteristics could indicate the presence of a trojan. The main difficulty of this method is that the deviations caused by small trojans might well be smaller than those caused by process noise and could therefore result in false negatives. An alternative is to try and activate any trojans on purpose. As the trojans mentioned in the literature often use rarely used nets as a trigger, the authors of [67] and [65] mention methods of purposefully activating such nets in order to activate possible trojans.

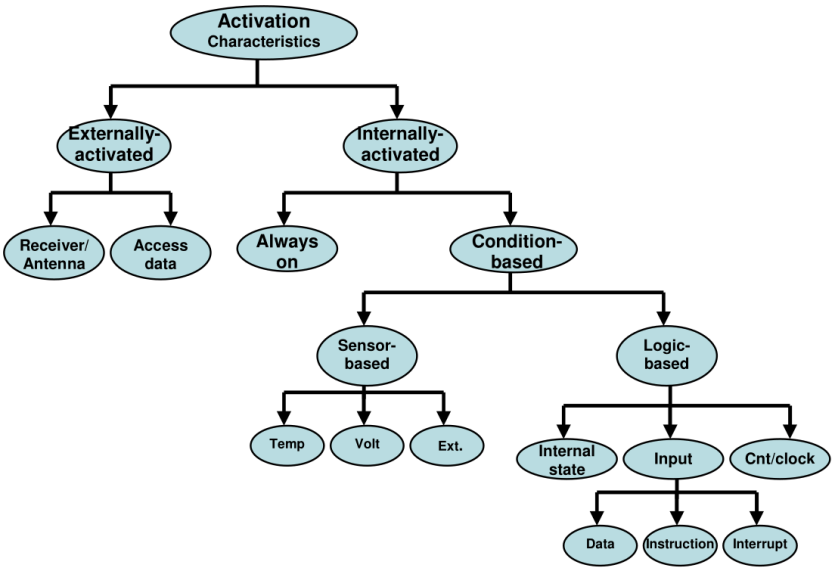


Figure 3.10: Potential hardware trojan activation methods [66]

The same papers mention other countermeasures during different phases of development as well, so the reader is referred to those for a more complete coverage of the subject.

4

Embedded Memory Security Module

It should now be clear that Internet of Things devices are an attractive target for attackers of all levels. To defend against attacks, strong defences are needed. At the software level, this means applications should be designed well and tested thoroughly before being used in the field. This alone, however, is not enough. If an attacker can tamper with data and applications at any stage, the device can be compromised. As such, the integrity, authenticity and confidentiality of data and applications must be ensured. This means that the installation and updating of applications must be protected, that the integrity of software can be verified before execution and that the physical storage media are protected against tampering. This chapter will first outline the unique environment that IoT devices present and then put the focus on hardware oriented memory protection in Section 4.1. Next, three variants of the Embedded Memory Security (EMS) module are presented in Section 4.2, as a solution against hardware attacks on external memory modules. Section 4.3 then covers relevant security considerations to make the presented module more secure. Finally, Section 4.4 proposes some additional extensions to build on top of the EMS module, to also protect the IoT device from software and network based attacks.

4.1. Concept

This section will briefly describe the unique situation that IoT devices find themselves in, and explain the reasoning behind focusing on hardware security. Section 4.1.1 first re-iterates the need for strong security in IoT devices. Next, Section 4.1.2 covers the restricted environments of these devices. Finally, Section 4.1.3 underlines why hardware security is particularly important for the Internet of Things.

4.1.1. Current Situation

Chapter 3 has shown that Internet of Things devices are vulnerable to attacks from all directions. Once a device is connected to a network, it is at risk of automated network based attacks targeted at any new device that comes online. More high-profile devices in important locations may be targeted directly. In that case, an attacker could spend the time and effort required to find vulnerabilities in the specific software running on those devices. Finally, those used in applications from the high-risk category defined in Chapter 2, run a real risk of being targeted physically by a powerful adversary. Unlike personal gadgets such as laptops and smartphones, IoT devices are spread out in the field and easily accessible as demonstrated in Figure 4.1. There is no user constantly nearby who notices when the device has been taken or tampered with, and as such an attacker may perform all sorts of attacks directly on the hardware. Ensuring the integrity and authenticity of applications running on, as well as the availability and confidentiality of the data collected by the IoT device is of primary concern. Especially since devices from the high-risk category may control parts of some critical infrastructure, their compromise may have dire consequences. Protecting these devices begins at the hardware level.

Many of the covered existing security measures are only focused on one type of attack, while leaving other parts unprotected. Anti-virus packages may protect against some software threats, but do not defend against network based brute-forcing and certainly not against anything hardware based. Similarly, countermeasures that detect attempts at hardware fault injection in the processor will not protect

Figure 4.1: Remote sensor node of CSIROs Fleck sensor network, installed in a field [68]. Cropped to fit page.

against malicious software leaking secret information, nor attacks on external memory or its interfaces. The same problem is present with more advanced security schemes such as Arm's TrustZone [12] technology. As covered in more detail in Section 3.2.3, it uses hardware barriers to set up a secure software environment which allows secure execution of selected applications, as well as secure firmware updates. It does not however protect against hardware tampering. Similarly, Maxim's MAXQ1061 [69] chip can protect a microcontroller's boot process against tampering on externally stored firmware, but it cannot secure external RAM of a device. Though implementing multiple of the mentioned measures at the same time may provide the required protection, it is not an elegant solution [70]: The combined performance penalties could be unacceptable for the (very) limited resources available in an IoT device. Furthermore, doing so could cover some attack vectors redundantly while others are left insufficiently protected. Security now also depends on the technology of multiple sources designed for multiple applications, and may fail entirely if one proves to have a breach. Ideally there would be a single solution that starts at the hardware level, is specifically aimed at IoT devices and covers all three attack vectors.

4.1.2. IoT Restrictions

By the nature of IoT devices they provide a rather restricted environment, a property that can be used to develop a security scheme. From the network point of view, things can easily be closed down substantially: Services and protocols that are not required for functionality may be disabled completely to prevent abuse. The few interfaces that remain can then be tightly controlled with strong authentication. The same is true for the software running on the device: The manufacturer has full control over what OS and applications come installed on it. They can limit that to what is needed to function while leaving everything else out, reducing the amount of possible bugs and weak points to exploit. They control when and what applications are updated and also what settings, if any, are available to the end user. Furthermore users may be prevented from installing other software, largely reducing the risks associated with viruses and trojans. Finally, the available hardware is rather limited. As IoT devices are meant to operate largely autonomously there is no need to include interfaces for keyboards or monitors, making it harder to mess with and observe the results compared to a regular computer. Similarly, debug ports such as UART and JTAG used during development can be left out on production models, or permanently disabled by burning on-chip security fuses after initial programming. Again,

any unneeded interfaces can be disabled to minimize the attack surface of the device. Then, the only interfaces left accessible are those used for sensors or actuators, a network interface and the PCB traces between the central processor and off-chip memory modules.

4.1.3. Goals

The goal of this thesis will be put at protecting IoT devices from physical threats, focused on the memory interfaces to off-chip RAM and storage. The reason for this is threefold: First, because of the unique restrictions of IoT devices, the attack surface from the network and software sides can be severely reduced with 'well written' software and strong authentication. Second, because of the way IoT devices are spread out in the field, constant supervision is costly and physical attacks are likely. Third, once an attacker has physical access to storage media, the system is vulnerable despite strong network and software defences. For example, once applications are running and considered trusted, an attacker may try to alter their execution through fault and code injection attacks. Similarly, he may flash custom or older firmware with known vulnerabilities to the device, or even swap the entire memory chip along with its contents with one from another device. Furthermore, consider the case where security settings are stored and either enabled or disabled based on some bits in memory. If these bits can be identified and flipped, the system is put at risk. Finally, sensitive information such as sensor data, cryptographic keys or proprietary algorithms may be stored in either flash or RAM, which could be leaked through physical access. Any software based security mechanisms are useless once an attacker can change the behaviour of 'trusted' applications through hardware. As such, attempts at influencing the code or execution of an application must be detected, and if so, execution must then be prevented or stopped. To achieve this, the next section will present a solution based around integrity verification and encryption of the contents of attached memory modules as the main contribution of this thesis.

4.2. The Embedded Memory Security Module

A typical setup of a processor with external memory modules is schematically shown in Figure 4.2. When the processor needs to write something to memory, it sends a request along with the data and its intended address to the memory controller. The controller then in turn sends a request to the memory module and provides the data over the memory interface. Once the data has been written successfully, the memory signals the controller that it is ready, which in turn forwards the signal to the processor, ending the request. Memory reads follow a similar method: The processor first sends a request for data from a certain address to the memory controller. It then forwards the request over the memory interface to the external module. When the request was processed and the data is available, the module provides the data along with a ready signal to the controller. The controller then simply forwards the data to the processor on its interface along with a ready signal, finishing the request.

When an attacker can read data from the memory of a device, its confidentiality is breached. Similarly, if an attacker can write data to memory, its integrity and authenticity is breached. As shown in Figure 4.2, once an attacker has physical access to a device, there are several ways to do just that by probing the traces to, or messing with the contents of the memory chip. To combat both these concerns, this thesis presents the Embedded Memory Security (EMS) module as a solution.

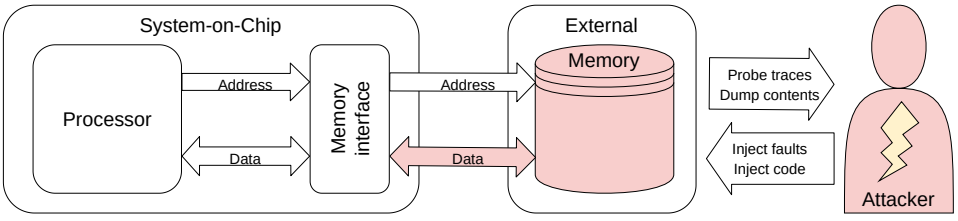


Figure 4.2: Schematic setup of processor with external memory, multiple attacks are possible

The EMS module is a hardware block that is integrated in the same chip as the main processor of a device. It sits on-die, between the processor and its memory controller as shown schematically in Figure 4.3. To both the processor and the external memory, the module is fully transparent; They have no control over the module in any way and are not even aware of its existence. As far as the processor is concerned, it is connected directly to the memory controller and nothing has changed interface-wise.

Since data is written to and read from memory on a per-address basis and each address in a memory device contains a block of data of fixed length, these blocks can be secured individually. The EMS module does this by either calculating and storing a hash, or more specifically a Message Authentication Code (MAC), for each data block to ensure integrity and authenticity, encrypting each block to ensure confidentiality, or both, depending on the required security and acceptable costs. Hashing and encrypting is done using a secret key, unique to each device, that is stored in hardware only accessible by the EMS module. This prevents it from being exposed by malicious software. The module can be used to secure any memory attached to the device, from RAM to flash, after which anything stored on it is bound to this particular device.

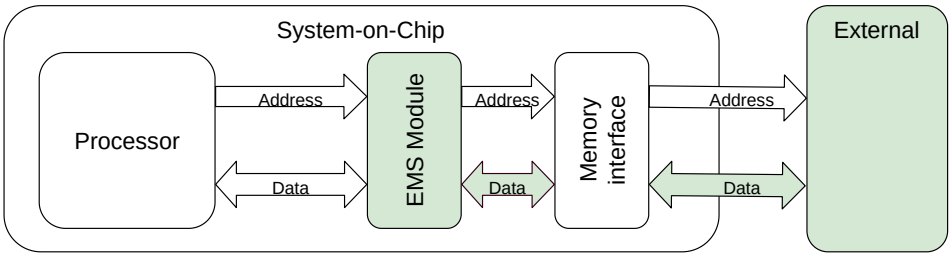


Figure 4.3: Schematic setup of processor with the EMS module, external memory is now secured

This section will cover the three variants of the EMS module: Those that ensure integrity and authenticity, those that ensure confidentiality and those that do both, in Sections 4.2.1, 4.2.2 and 4.2.3 respectively. There, the concept and workings of each version will be explained in detail. Finally, Section 4.2.4 discusses expected security and performance figures for each implementation.

4.2.1. Integrity and Authenticity

The first variant of the EMS module is intended to ensure the integrity and authenticity of code and data stored on a device. This has the highest priority as it does not only protect the device itself, but also the systems it belongs to: If no unauthorized software is able to run and no data can be tampered with, systems depending on the device's information are protected. This verification is done by calculating and storing a MAC of each block of data as it is written to external memory. This MAC is then verified when the data is read back. Verification is done on a per-address basis.

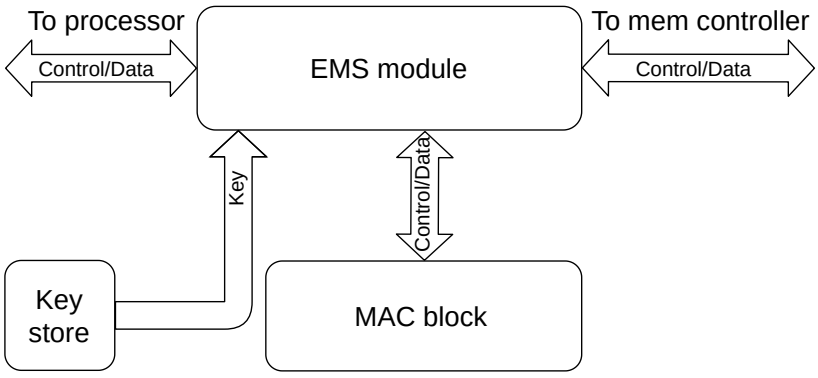


Figure 4.4: Schematic overview of components and interfaces of the hashing EMS module

From the processor and memory controller point of view, the EMS module is transparent with the exception of an additional error flag towards the processor. It has access to a keyed hashing block that provides the hardware function to calculate the MAC over a given message, as well as a secure cryptographic key store. When writing data to memory, calculated MACs are stored together with the original data. Upon reading, the MAC is calculated again over the read data and compared to the value stored in memory. If they are the same, it is verified that the data was generated by the device itself and that it was not altered, hence ensuring its authenticity and integrity. If they differ however, this is immediately detected by the EMS module and additional security measures may be triggered by

setting its error flag. A schematic overview of the hashing variant of the EMS module and its interfaces is shown in Figure 4.4.

Writing to Memory

This variant of the EMS module requires an additional location in memory to store its MAC tags for each block of data. When the processor requests to write data to a certain address in memory, it first calculates a MAC over this data and determines the address to store it. After this, both the data and its tag are sent to the memory controller to be written to memory at their respective addresses. Schematically this process is shown in Figure 4.5. The steps taken by the module upon receiving a write request are as follows:

1. EMS module receives a write request along with data and its address from the processor
2. EMS module sends data and key to MAC block to process
3. EMS module receives the calculated MAC from the MAC block once it is done
4. EMS module sends data and address to memory controller to write
5. EMS module receives ready signal from controller once write is complete
6. EMS module sends MAC and its address to memory controller to write
7. EMS module receives ready signal from controller once write is complete
8. EMS module sends ready signal to the processor, completing write request

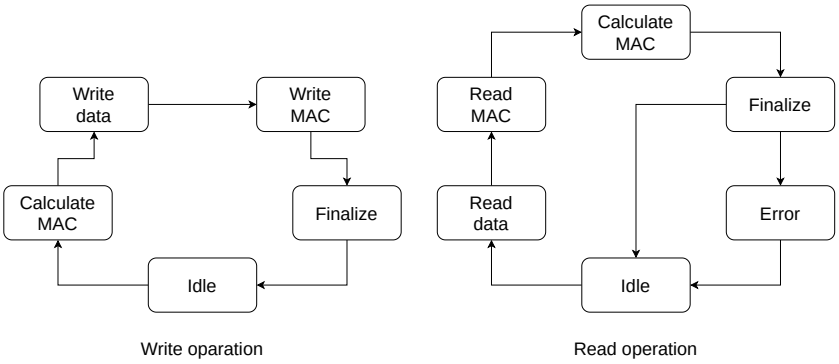


Figure 4.5: Schematic overview of the MAC state machine

Reading from Memory

Similar to the write request, on read requests the EMS module needs to read both the data and its corresponding MAC from memory. It then calculates the MAC of the data and compares it to the tag that was stored along with it. If the tags match, the data's integrity and authenticity has been verified and the data is provided to the processor. If not, the module raises an error flag to signal that something is wrong. This process is shown schematically in Figure 4.5. The steps taken by the module after receiving a read request are as follows:

1. EMS module receives a read request along with its address from the processor
2. EMS module sends data address to memory controller to read
3. EMS module receives ready and data from the controller once read is complete
4. EMS module sends MAC address to memory controller to read
5. EMS module receives ready and MAC from the controller once read is complete
6. EMS module sends data and key to MAC block to process
7. EMS module receives the calculated MAC from the MAC block once it is done
8. EMS module sends data and ready to the processor if MACs match or raises flag if not, completing read request

4.2.2. Confidentiality

The second variant of the EMS module is intended to ensure the confidentiality of code and data on the device. Confidentiality is relevant if a device handles information or applications that are to be kept secret from adversaries. The module provides this functionality by encrypting all data leaving the central processor toward its external memory modules, and decrypting all data coming back.

Assuming the cipher and its implementation are secure, encrypting all traffic flowing between a processor and its external storage media will prevent an attacker from gaining access to secret information through hardware attacks. Intercepting the traffic to, or dumping the contents of the storage media will only show encrypted data. Furthermore, memory based fault injection attacks as shown in Chapter 3 become impractical as the adversary has no way of knowing what data was actually changed. This is extended by the fact that any output back to memory, which may or may not be a result from the attack, is also encrypted and as such unreadable. Finally, it prevents any types of hardware based code injection as without knowing the encryption key, anything written to memory will become garbled by decryption before it reaches the processor. Code injection here becomes nothing more than fault injection with a larger amount of random data.

The EMS module is fully transparent to both the processor and its memory controller. This variant has access to a cipher block and a secure cryptographic key store. This cipher block provides the function to encrypt a plaintext message with the secret key, as well as to decrypt a ciphertext message. A schematic overview of the encryption variant of the EMS module and its interfaces is shown in Figure 4.6. This variant of the module encrypts and decrypts each block of data corresponding to a single address individually as it is transferred between the processor and its memory.

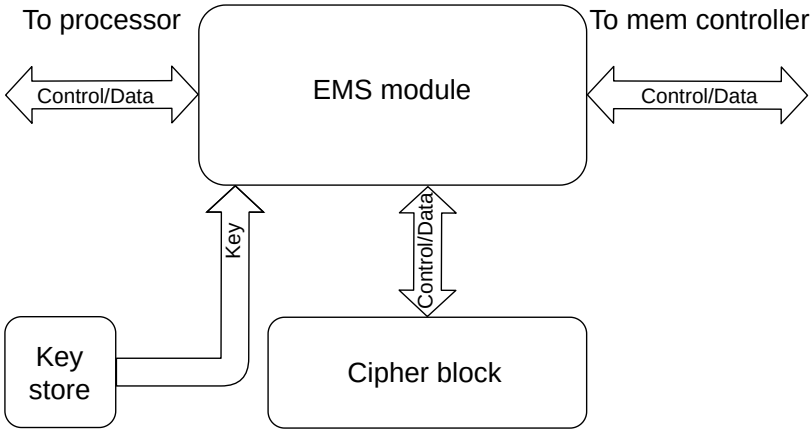


Figure 4.6: Schematic overview of components and interfaces of the encryption module

Writing to Memory

The ciphertext messages have the same size as the plaintext data, meaning this variant of the EMS module does not have additional memory requirements. When the processor issues a write request to write data to memory, the module encrypts the data and then forwards it to the memory controller. This process is shown schematically in Figure 4.7. The steps taken by the module upon receiving the request are as follows:

1. EMS module receives a write request along with plaintext data and its address from the processor
2. EMS module sends the plaintext data and key to the cipher block to encrypt
3. EMS module receives ciphertext data from cipher block once it is done
4. EMS module sends ciphertext data and its address to the memory controller to write
5. EMS module receives ready signal from controller once write is complete
6. EMS module sends ready signal to the processor, completing write request

Reading from Memory

Similar to the write requests, the module also receives the requests from the processor to read data from memory. Once it then receives the encrypted data from the memory controller, it decrypts it and

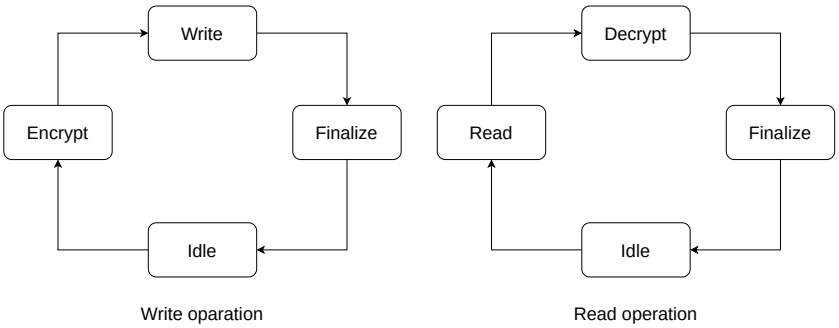


Figure 4.7: Schematic overview of encrypt state machine

passes it on the the processor. This, too, is shown schematically in Figure 4.7. The steps taken by the module upon receiving the read request are as follows:

1. EMS module receives a read request along with its address from the processor
2. EMS module sends the address to the memory controller to read
3. EMS module receives ready and ciphertext data from the controller once the read is complete
4. EMS module sends ciphertext data and key to the cipher block to decrypt
5. EMS module plaintext data from the cipher block once it is done
6. EMS module sends plaintext data and ready to the processor, completing read request

4.2.3. Integrity, Authenticity and Confidentiality

The third variant of the EMS module is meant to ensure the integrity, authenticity and confidentiality of the code and data stored on the device to achieve the highest level of security. This is done by combining the functionality of the first two variants and both encrypt the data and verify a MAC. Because the contents of external memory are encrypted, an attacker will not gain access to the data by intercepting traffic to the memory chips, or by dumping their content. Furthermore, it will prevent attacker from injecting fault or code and observing the results. The MAC additionally ensures that any such attempts are also detected by the processor, in order to activate security measures.

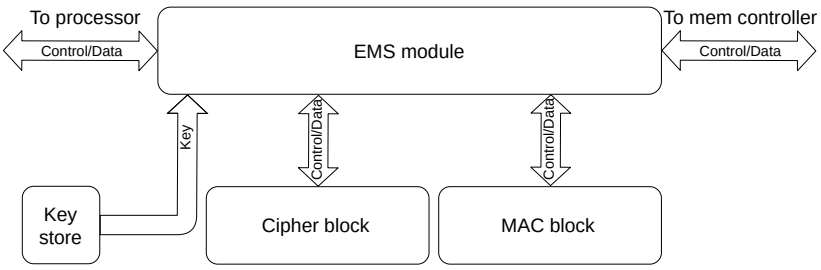


Figure 4.8: Schematic overview of components and interfaces of the combined encryption and hashing module

Again, the module has the same transparent interfaces between the processor and memory controller, with the addition of an error flag. It has access to both a cipher block and keyed hashing block, as well as a secure cryptographic key store. The cipher block provides the function to encrypt and decrypt data with the secret key, and the keyed hash function can calculate a MAC over a message with its key. When writing data, the data is encrypted first using the cipher. A MAC is then calculated over the ciphertext, and both are stored in memory. On reads, both the ciphertext data and its MAC are read from memory. The mac is calculated again over the ciphertext and compared with the stored MAC. If they match, it is verified that the data was not altered and the ciphertext is decrypted. If they do not, the error flag is set towards the processor. A schematic overview of this variant of the EMS module and its interfaces is shown in Figure 4.8.

Writing to Memory
As mentioned above, this variant of the EMS module performs a number of steps when writing data to memory. As a MAC is stored with the data, an additional spot in memory is required to store it. When the processor requests to write data to a certain address in memory, the data is first encrypted. It then calculates a MAC over the ciphertext, and determines the address to store it. After this, both the encrypted data and its tag are sent to the memory controller to be written to memory at their respective addresses. This process is displayed schematically in Figure 4.9. The steps taken by the module upon receiving a write request are as follows:

1. EMS module receives a write request along with plaintext data and its address from the processor
2. EMS module sends the plaintext data and key to the cipher block to encrypt
3. EMS module receives ciphertext from the cipher block once it is done
4. EMS module sends ciphertext and key to MAC block to porcess
5. EMS module receives the calculated MAC from the MAC block once it is done
6. EMS module sends ciphertext data and its address to the memory controller to write
7. EMS module receives ready signal from controller once write is complete
8. EMS module send MAC and its address to the memory controller to write
9. EMS module receives ready signal from controller once write is complete
10. EMS module sends ready signal to the processor, completing write request

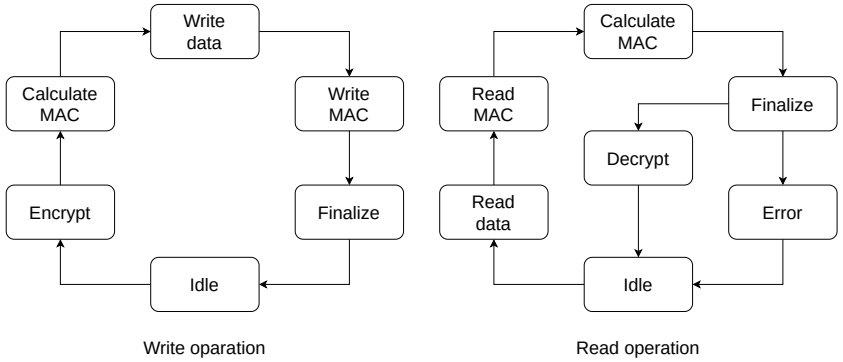


Figure 4.9: Schematic overview of the encrypt + MAC state machine

Reading from Memory
Reading from memory again follows a similar process as before. Besides the encrypted data itself, the module also needs to read the corresponding MAC from memory when it receives a read request from the processor. Once it has received both, it calculates the MAC of the ciphertext data and compares it to the tag that was stored along with it. If the tags match, the data is considered to be authenticated and the ciphertext is provided to the cipherblock to be decrypted. If not, the module raises an error flag to signal that something is wrong. After decryption is done, the plaintext data is provided to the processor. Once more, this process is displayed schematically in Figure 4.9. The steps taken by the module when receiving a read request are as follows:

1. EMS module receives a read request along with its address from the processor
2. EMS module sends data address to the memory controller to read
3. EMS module receives ready and ciphertext data from the controller once read is complete
4. EMS module sends MAC address to the memory controller to read
5. EMS module receives ready and MAC from the controller once read is complete
6. EMS module sends ciphertext data and key to the MAC block to process
7. EMS module receives the calculated MAC from the MAC block once it is done
8. EMS module sends ciphertext data and key to the cipher block to decrypt if MACs match or raises flag if not

9. EMS module receives plaintext data from the cipher block once it is done
10. EMS module sends plaintext data and ready to the processor, completing read request

4.2.4. Expectations

Now that the three variants of the Embedded Security Module module have been presented, some points will be made regarding their levels of protection, their performance impact and finally some trade-offs when implementing them on different types of memory.

Security

The first variant, with MACs, is intended to ensure the integrity and authenticity of code and data. Without knowing the key, the attacker cannot generate matching MACs to any injected faults or custom code, meaning the module will detect any such attempts. By its nature however, hashing does not protect the confidentiality of any stored data; an attacker may still be able to intercept and read it without raising alarms.

The second variant of the module, with encryption, is intended to ensure confidentiality of code and data. It has the beneficial side-effect of making some types of attack more difficult if not impossible to perform successfully. Without knowing the key, an attacker does not know what data is changed by injecting faults and can not insert his own code. However, it will not prevent the attacks from happening in the first place. The processor may still be made to perform unintended random actions, even though the attacker can not directly control what happens. When one bit is changed in memory that stores application instructions, the instructions are likely to be completely different if not illegal once they reach the processor. This would cause applications or the whole operating system to crash.

The third variant, with both encryption and MACs, is intended to ensure all three; confidentiality, integrity and authenticity. It combines the security aspects of both. Without knowing the key, an attacker can not read any secret information, nor can he perform any kind of injection attack without detection. This third variant as such offers the highest level of protection. The levels of protection of each variant against some hardware attacks is shown in Table 4.1.

Table 4.1: Protection offered by variants of EMS module against hardware attacks

	Protected against attack type:				
Module variant:	Stealing data	Fault injection	Code injection	Side channel	Replay
MAC	No	Yes	Yes	Depends	No
Encrypt	Yes	Partially	Yes	Depends	No
Encrypt + MAC	Yes	Yes	Yes	Depends	No

Aside from the mentioned attacks based around stealing data or injecting faults or malicious code, there are other hardware attacks as mentioned in Chapter 3. Specifically side-channel analysis is a type of attack that the EMS module does not inherently protect against. Depending on the implementation, it might be possible for an attacker to gain information about the secret key by observing power consumption or other metrics. Securing the system against these types of attacks is entirely implementation and technology dependent and many countermeasures already exist. Defending against side-channel analysis is therefore considered out of scope for this project.

Another inherent weakness of transparent cryptography is the vulnerability to replay attacks. These are attacks where an adversary replaces a message with an earlier message, without the sender and receiver noticing anything has happened. It comes from the fact that without additional measures, a specific plaintext with a specific key always leads to a specific ciphertext. The same holds true for MACs; when the same message is hashed with the same key, it will lead to the same MAC. To abuse this fact, an attacker would have to save the messages (As well as their MAC, if used) and could then replace data in memory with older versions (Along with their MAC), and the module would not detect anything wrong. It could be argued that encryption will prevent an attacker from knowing what data he replaced with what other data, making the attack impractical. Similarly, hashing would require an attacker to save and replace two values in memory, making the attack more difficult to perform. However, the modules on their own do not offer specific protection against this type of attack. Some potential additions to the modules to include this protection will be covered in Section 4.3.

Performance
Adding security measures comes at a cost. Aside from the area requirement on-die of the EMS module, the additional calculations will take time. This time will be added directly to the time it already takes to transfer data to and from memory.

The exact time it takes to encrypt or decrypt the data for a single memory access is implementation specific. It depends on how fast the selected cipher can process the amount of data that the platform transfers in one go. Table 4.2 shows some typical access times for different memories. From this data it is clear that an access to external DRAM modules can easily take more than a hundred clock cycles of the processor, while flash for storage can take many thousands of cycles per access. As such, an additional delay of, say, ten extra cycles for encryption would not significantly affect the memory performance.

Table 4.2: Access times for different memory technologies. Data from [71], page 378

Memory technology	Typical access time
SRAM semiconductor memory	0.5-2.5 ns
DRAM semiconductor memory	50-70 ns
Flash semiconductor memory	5.000-50.000 ns
Magnetic disk	5.000.000-20.000.000 ns

The MAC variants of the EMS module are expected to have a much larger performance impact. Aside from the time it takes to calculate the hash value, this value will also need to be stored. Doing so will require an additional memory access per access, in order to read or write the MAC. This will effectively double the time spent on memory transfers. Furthermore, this will require additional memory space to store the hash. The exact amount depends on the size of the hash relative to the input data. The upper bound to this would be when the hash is the same size as the input data, as using hashes that are larger do not make much sense. This would require a doubling of the installed memory to have the same amount of system resources available. A smaller hash would reduce the amount of memory required to store it, potentially at the cost of security. No matter the size however, an additional memory access will always be necessary.

In the absolute worst case scenario this would lead to memory performance being halved, while requiring twice as much of it. In practice this is not expected to be realistic. Real-world performance will depend greatly on the availability, size and performance of any caches near the processor, as well as the memory behaviour of the running applications. This real-world performance will be explored later in Chapter 6.

Memory Options
The EMS module is a intended to be a modular block. Its interfaces can be altered to fit any type of memory interface. This means it can be implemented to secure external flash storage, as well as the RAM on higher-performance IoT devices. Lightweight sensor nodes such at the Tmote Sky may not have external RAM to secure, but even these devices can have external flash chips to store additional code and data [19]. As such, the EMS module is applicable to a wide range of devices and applications.

Securing external RAM modules will protect applications and their data while they are running. By hashing the memory, it can be ensured that an attacker can not change any security parameters. Furthermore, he will not be able to modify the execution of applications without being detected. Encryption would hide any secrets that are temporarily stored, such as cryptographic keys and possibly the operations of some proprietary algorithms.

Similarly, securing external flash can protect code and data while it is at rest. Unlike RAM, this kind of memory is persistent; It will remain the same when the device is powered off. An attacker with access to the device could dump the contents and take his time to explore it, change data or even swap the entire chip for another one. Hashing it ensures that the stored applications are authorized by the device and haven't been tampered with, meaning they can be trusted to execute the way they're supposed to. Encrypting it ensures that an attacker can not get access to secret or proprietary code and data. In both cases, if an attacker breaks anything, the device is practically bricked. Though this could also be seen as a valid attack vector to disrupt some operations, it should be noted that since the attacker would have physical access anyway, actually bashing it with a real brick would be more efficient.

4.3. Security Considerations

When implementing the modules proposed above, there are several additional steps and security measures that should be considered. These may be basic requirements to secure the systems that use the modules, as well as potential options to counter some other types of attacks. This section will cover some of these considerations next.

4.3.1. Key Handling

The keys used by the Embedded Memory Security modules to perform MACs and encryption must be kept secret. If these keys were to be leaked, the security of the system can no longer be guaranteed. An attacker would be able to read any data stored in the device, as well as encrypt any message of his choosing and generate matching MACs. At this point, it would be as if the EMS modules were not there at all. To prevent this, these keys must never be allowed to leave the device or be readable by any software running on it. Furthermore, if both MACs and encryption are performed, both operations should use a separate key. If the same key is used for both, some information about this key may be leaked through interactions between the ciphertext and its MAC [25]. Furthermore, if for example the key is somehow recovered from a broken MAC scheme, when the same key is used the confidentiality of the data is now also lost. To extend these requirements even further; a sufficiently determined (and funded) attacker may still be able to extract a device's keys through invasive hardware attacks. Though such an attack would render that particular device inoperable, the security of any other devices that utilize the same keys should now be considered broken. To combat this, each device coming from the factory must have its own unique keys that never leave the device.

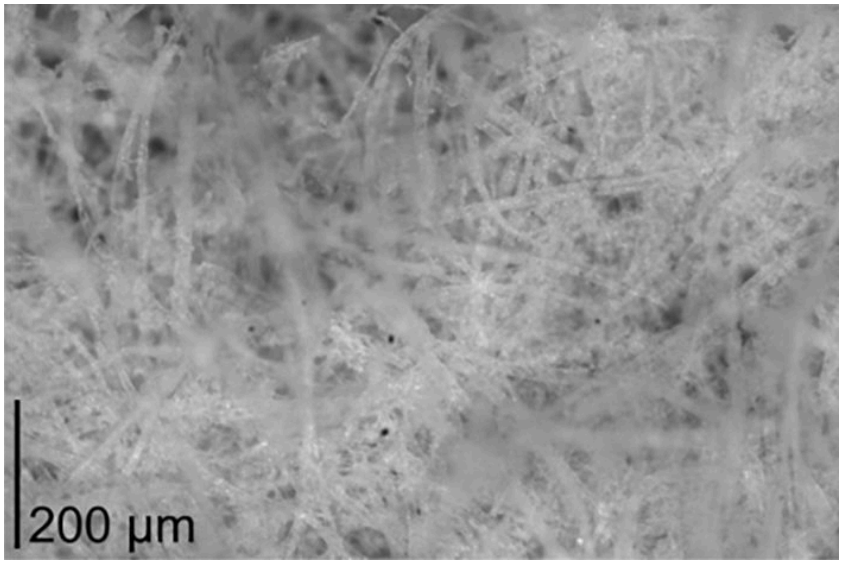


Figure 4.10: Unique fibre layout of a piece of paper, visualising the concept of a PUF [72]

Implementing a Physical Unclonable Function (PUF) in each device is one way to generate device-specific keys. The main concept of a PUF is to use any manufacturing variations inherent in small-scale electrical circuits, to generate unique 'fingerprints' for each device. These manufacturing variations are random, and cannot be identically reproduced intentionally. A visual example of this concept is shown in Figure 4.10, where the exact fibre structure of a piece of paper represents the random manufacturing variations. Reproducing this exact structure intentionally in another piece of paper would be infeasible. In the context of electronics these variations could be found in parasitic capacitances, gate leakage of transistors, wire delays etcetera. This allows for building a hardware function that generates certain outputs (responses) to given inputs (challenges), unique to each device and impossible to clone or predict [73]. There are two main categories of PUFs; weak and strong, depending on their amount of challenge-response pairs. A weak PUF only support one to a 'few' challenge-response pairs. These may be used to generate a single unique key for a device. Strong PUF's on the other hand support many challenge-response pairs. When such pairs are recorded and stored right after manufacturing, they can be used to authenticate a device later. An advantage of these is that even if an attacker

manages to record some pairs, he will still not be able to extrapolate the response for any other input and as such cannot perfectly impersonate a device. A more in-depth discussion about the differences is covered by Holcomb et al. in [73]. For a complete overview about the properties and various methods of constructing a PUF, the reader is referred to the taxonomy of McGrath et al. [72]. By adding a PUF, it can be ensured that each device will have its own unique secret key.

4.3.2. Authentication Methods

The presented EMS modules implement a MAC function for authentication and encryption for confidentiality. When both are required, an 'encrypt-then-MAC' scheme is used as discussed in Section 4.2. Other alternatives do exist, such as 'encrypt-and-MAC' where the MAC is calculated over the plaintext and 'MAC-then-encrypt', where the MAC is also calculated over the plaintext but then encrypted along with it. These alternatives however are not 'generically' secure and put more constraints on the type of cipher and MAC function to use [25]. Bellare et al. discuss the security properties of these three schemes in more detail, and provide security proofs in [74].

Aside from these schemes that combine a cipher and a MAC function, there are other methods to perform authenticated encryption. Several modes of operations have been developed for blockciphers that provide this functionality, including Offset CodeBook (OCB) mode [75], Counter with cipher block Chaining Mode (CCM) [76] and the Galois/Counter Mode (GCM) [77]. OCB was submitted to be included in the IEEE 802.11i standard, but due to patents and licensing issues it was refused. CCM was developed as an open alternative for this purpose and is currently part of said standard. Finally, GCM was developed as a high-throughput alternative to other authenticated encryption schemes, by being parallelizable in hardware implementations. It is defined in NIST standard SP 800-38D. An example of the GCM encryption process is shown in Figure 4.11.

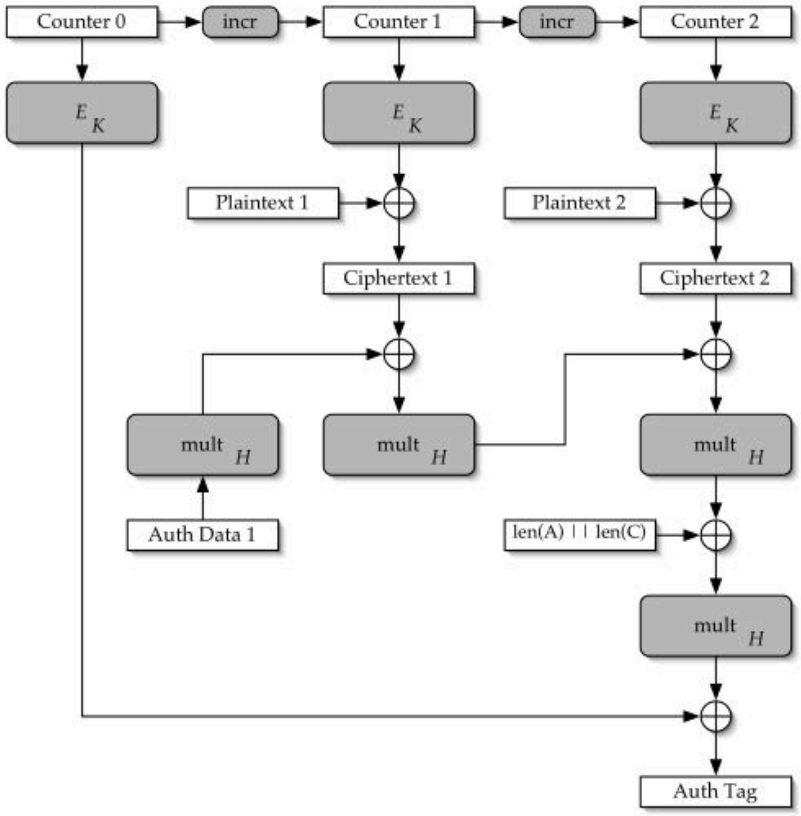


Figure 4.11: Encryption process of GCM mode, encrypting two plaintext blocks and using one block of authentication data [77]

These modes, though interesting, were not further explored for the application of this thesis for two main reasons: First, the EMS modules are intended to be modular, meaning it should be possible to only encrypt or only MAC as well, only implementing the required hardware functions. This would defeat the purpose of these authenticated encryption modes. Secondly, these modes of operation are

not particularly light-weight compared to the alternatives. They all require blockciphers that use blocks of at least 128-bits, which would disqualify most if not all of the purposely-designed lightweight ciphers covered later in Chapter 5. For completeness however, one GCM implementation was synthesized in Chapter 6 as a comparison to the results of this thesis.

4.3.3. Memory Vulnerabilities

For the modules that support encryption, one potential security issue that remains comes from patterns in memory being visible to an attacker. Each individual block of data that gets encrypted is only the size of data stored at one address in memory, i.e. 128 bits. This introduces an issue similar to using a block cipher in the basic Electronic Code Book (ECB) mode of operation to encrypt a large file: As each block of data is encrypted with the same key, this means identical plaintexts lead to identical ciphertexts in memory. Though the attacker may not know what the plaintext data is, he may still be able to recover some information. An example of this issue is shown in Figure 4.12, where an image of the Linux mascot Tux was encrypted with AES-128 in ECB mode. Note how though the image is encrypted, its outline is still clearly visible. Whether or not this is an issue depends on the actual data, as it may allow an attacker to deduce the plaintext of common values.

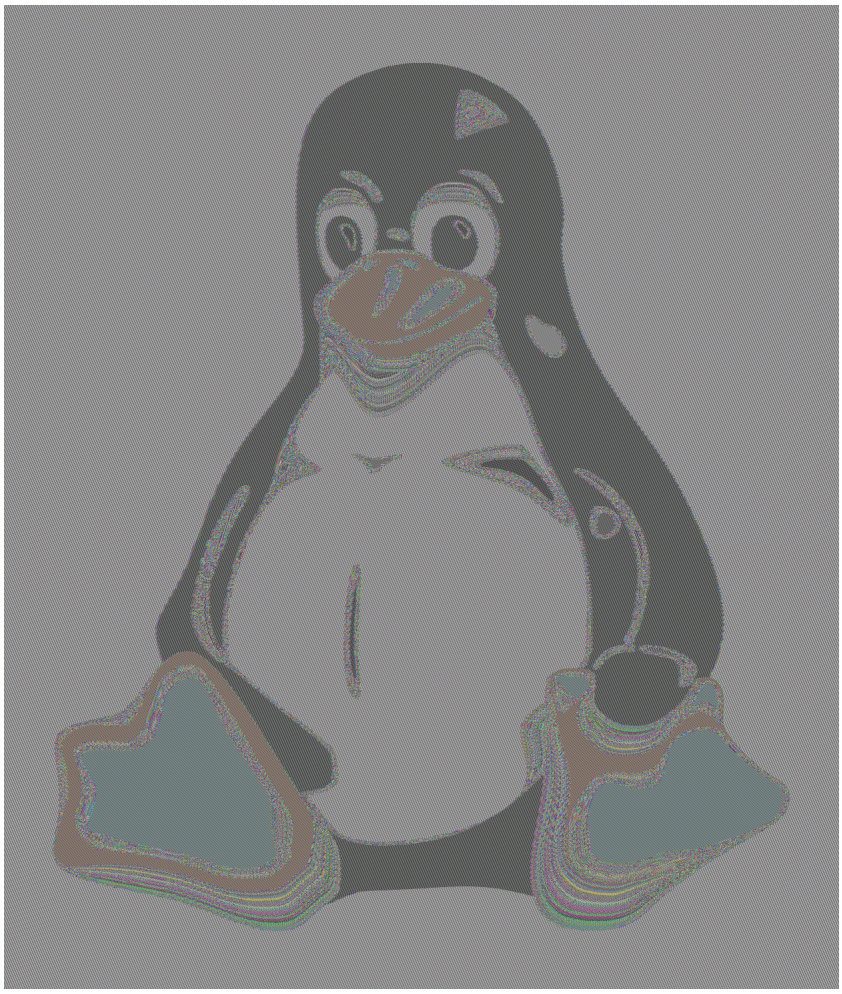


Figure 4.12: When encrypting each block individually with the same key, some details of the original data may still be visible [78]

Modes of operation are meant to combat the problem of encrypting data larger than a single block size. They are methods to manipulate the inputs or outputs of a block cipher, based for example on the results from previous blocks, in order to ensure that identical plaintext blocks lead to different ciphertext outputs. An overview of different modes of operation and their properties is given in [25]. In our application however, this method is not applicable: Here the issue is not that the plaintext contains multiple identical blocks, but that the entire (small) plaintext will occur multiple times. As such, a different

approach is needed to let the same plaintext result in a different ciphertext at different addresses.

A similar issue are the replay attacks mentioned before. They are similar in that an attacker can simply observe the contents of the memory to get the needed information. He can then replace any valid data-MAC pair with another one that was stored earlier or at a different address, without being detected by the EMS modules. This issue is solved in some network protocols by including session-IDs or time-stamp information, which are then transmitted along with the data and verified by the recipient. Both of these would need to be generated and stored alongside the data and MAC however, further increasing the memory footprint of the EMS module. Unlike data transferred over the network where packets arrive shortly after being sent, the moment when data is read back after being written to memory is unknown and may take a long time. This makes it difficult to verify if data was rolled back by a replay attack without storing. Encryption reduces the risk of replay attacks being effective as the attacker does not know what data is being replaced with what, but it does not prevent them from slipping detection.

Several options of including the address in the encryption algorithm were explored, most of which were discarded as being too costly or out of scope of this thesis. The most straightforward solution would be to XOR the key or plaintext data directly with the address before encrypting. Though this would create different ciphertexts for identical plaintext messages, it would expose the system to related-key and related-plaintext attacks respectively and cannot be considered secure. Another option was to implement a Key Derivation Function (KDF) or use a strong PUF with the address and main key as inputs to produce more keys. Generating such a key each time an address is accessed would be costly however, and a search did not lead to any existing efficient hardware-targeted KDFs. As such this option was not explored further. Alternatively, a so-called tweakable cipher could be used. These take, aside from the key and message, an additional 'tweak' input to introduce variability. Here, the address could be used for this purpose. Unlike the key, this tweak does not need to be kept secret to remain secure, and no additional hardware is required aside from the cipher [79]. All of these solutions would solve the issue of patterns in memory and would prevent replay attacks with data from other locations. However, none would fully prevent them from a theoretical level, as an attacker may still roll back data to earlier versions from the same address. These solutions are mentioned for reference, and were not further implemented in this thesis.

4.4. Security Extensions

The proposed solution protects a device, its data and its software from hardware based attacks. However, as the module is completely transparent to the processor, it does nothing against any software based attacks. In particular, if an attacker can exploit some vulnerability in the installed applications or the update mechanism, the EMS module on its own will not help. Therefore, we propose the addition of two other strategies to leverage the security provided by EMS module. They are referred as *Cloud Protocol* and *Software Binding*.

4.4.1. Cloud Protocol

Ideally, an attacker would be unable to run any code on a target device. This would prevent him from leaking or manipulating any data, or controlling any attached machinery. To ensure that this is the case, all applications running on the device must be verified and checked for tampering at all stages. If some tampering has taken place it must be detected, and the application must not be allowed to be executed. From the IoT restrictions mentioned above, it follows that there is one single channel for updates that needs to be secured - the *Cloud to Device* communication. Updates may only come from the authorized server, and any tampering with data during transit must be detected immediately. To ensure this, some method of authentication and securely transferring the updates is required.

The concept here is to build a *Cloud-to-Device* protocol having the unique ID of the device as the root-of-trust. Only the manufacturer has access to the unique ID, generated or retrieved after the integrated circuit was created. Using such ID as part of the protocol, classical methods [80] can be used to build a resilient and unique communication between the Cloud and the Device. Additionally, by having a Physically Unclonable Function (PUF) [72], the protocol can take benefit of multiple unique IDs. In this case, the Cloud generates random values to be used as the PUF Challenge, and hence, improve the quality of the secure protocol.

4.4.2. Software Binding

Since the manufacturer has full control over what software should run on the device, these applications can be bound to the particular hardware. This prevents any proprietary software from running on an unexpected hardware platform (i.e., attacker's equipment), and also guarantees that if there is an application with hidden malicious functionality present (e.g., if that functionality was not detected during the tampering test), it can not spread its code among other devices.

The concept of software binding aims to link the applications to the target hardware platform. The operating system is not considered for binding, as it contains its own security mechanisms. In fact, the Operating System has an important role in the process of checking the applications and guaranteeing that only valid processes can run in the processor. To achieve such functionality, both Cloud and Device must know a secret of each other. The Cloud already has the unique IDs of each device in its database, but the Device must have also know a secret that refers to the Cloud. Such data can be stored in the Device after fabrication (in the same step where the IDs are generated/retrieved). Finally, the application itself must receive additional data or instructions that applies the secrets. Thereafter, before starting the application, the Operating System can check the content or run a specific part of the process to evaluate if that code matches with the hardware ID, and also if it is authentic.

5

Implementation

While developing the Embedded Memory Security module, several design choices had to be made in order to build a working prototype. The first decision was about the processor development platform, as that determines the required interfaces of the module. Next, both a cipher and MAC function had to be selected out of many options. This chapter will cover these design choices and tradeoffs, as well as the structure of the EMS modules themselves. To that end, Section 5.1 first discusses the platform, Section 5.3 covers the encryption algorithm and Section 5.2 goes over the MAC function. Finally, Section 5.4 covers the interfaces and variants of the EMS modules.

5.1. Test Platform

The developed EMS modules are not stand-alone units; they are meant to be integrated into an actual processor. Their interfaces and consequently their performance will depend largely on how, and in what processor they are added. As such, a (soft-core) test platform on an FPGA is required to simulate the real-world situation when implemented in actual hardware. Since the EMS modules are aimed at IoT implementations, this platform should be similar to those found in actual IoT devices. This means it should at least be able to run arbitrary programs and use on-board LEDs to signal various things.

Though several options were available online, project criteria and limitations reduced the selection size significantly. Some requirements were formulated in order to deal with this, which will be mentioned next. This is then followed by a list of available platforms and their trade-offs. After this, the specifications of the chosen platform will be covered in more detail.

5.1.1. Requirements

At the beginning of the project, several requirements were put on the test platform. These are based on the available development hardware and software, but also on keeping the project open to allow potential porting and further development at a later stage. The requirements are listed as follows:

- **Req 1:** The first requirement is that the test platform must be open source. This makes it possible to modify parts of the processor as needed to insert the new authentication module. Furthermore it allows for detailed debugging by analysing all internal signals in a simulator. As changes to the platform will have to be made, this is a hard requirement.
- **Req 2:** The second requirement is that the platform must have specifications comparable to those found in general IoT devices. As covered in Chapter 3, this varies widely from 8 bit microcontrollers to full fledged multicore processors. Since adding the proposed EMS module only makes sense when using external memory, the focus is put on higher end 32-bit microcontrollers or better. Ideally the platform has access to instruction and data caches to model their effects on the performance impact of the module.
- **Req 3:** The third requirement is that the platform must not be strongly tied to one single vendor of FPGA's by making extensive use of that vendor's custom IP blocks. The Computer Engineering department uses hardware from multiple manufacturers including Xilinx, Altera (Intel) and Lattice Semiconductors. As such, vendor lock-in should be avoided by not relying heavily on IP blocks

from one particular source if not absolutely required. Furthermore said custom IP blocks may not be open source, violating the first requirement.

- **Req 4:** The fourth requirement is that the platform must be able to fit and run on the available FPGA development boards for testing. For this project a fairly limited Digilent PYNQ-Z1 board is available of which a picture is shown in Figure 5.1. It contains a Xilinx ZYNQ XC7Z020-1CLG400C FPGA with 13.300 logic slices, 53200 LUTs and $630KB$ of block RAM. For more details, the reference manual of the board can be found at [81].

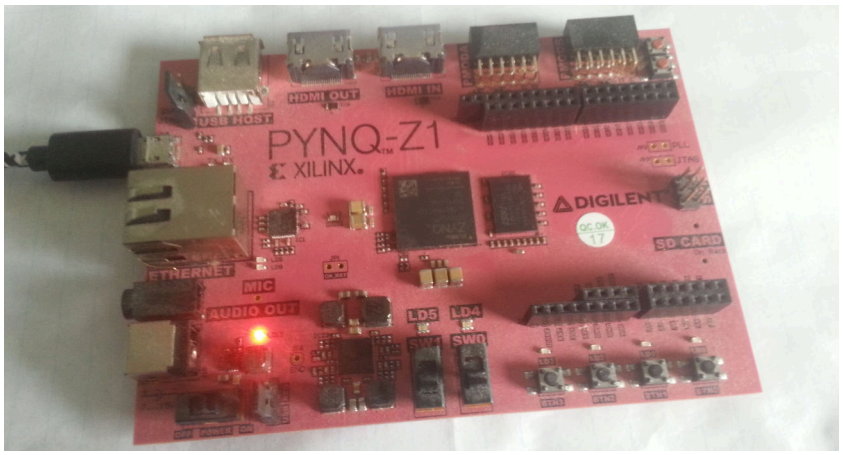


Figure 5.1: Picture of the available FPGA development board, a Xilinx PYNQ-Z1

5.1.2. Options

A search was performed online for available processor development platforms, which were then held against the aforementioned requirements. The first and third requirement disqualified several options that require a license to use or are tied to a specific company, examples being ARM cores and Xilinx's Microblaze platform. As per the second requirement the search was limited to relatively high performance platforms, which excluded many small 8-bit open microcontrollers from the list. Finally, the fourth requirement, based on the limitations of the development board, meant that any larger processors would not be able to be ran in hardware. One of these is the RISC-V Ariane processor that otherwise met all other requirements [82]. The resulting list of options that fulfilled at least some requirements is shown in Table 5.1. Here, the green colour indicates that the platform fulfils a requirement, red indicates that it does not, and yellow indicates a partial dependency that could be worked around with some effort. From this list there remain three options: The Ariane platform [83], the Pulpissimo platform [84] and a custom platform that is already in use by the computer engineering department. All three of these are based around a RISC-V core architecture.

Table 5.1: Requirement compliance for different test-platforms

Platform:	Architecture:	Req 1:	Req 2:	Req 3:	Req 4:	Notes:
Arm M4	ARM	No	32 bit uC	ARM	No	Not available
Ariane	RISC-V	Yes	Full processor	Xilinx*	No	64 bit, can run Linux
Microblaze	RISC/DLX	No	32 bit uC	Xilinx	Yes	Too vendor bound
Pulpissimo	RISC-V	Yes	32 bit uC	Xilinx*	Maybe	Similar to TU custom
TU custom	RISC-V	Yes	32 bit uC	None	Yes	Department support

Performance-wise the Ariane platform has the highest specifications [83]. It is a full-fledged 64-bit processor with multiple cache levels, support for external flash and DRAM, as well as a Memory Management Unit (MMU), making it able to run Linux. Especially this last property makes it an attractive development platform. However, it does use some Xilinx specific components in its AXI-bus and the developers only support the Digilent Genesys 2 FPGA board [85]. Furthermore, because the minimum single-core configuration already requires more than 85.000 LUTs of FPGA area, it would not fit in the FGPA on the available PYNQ Z1 board.

The Pulpissimo and the TU custom platforms are very similar. Both use the RI5CY core [86], which is a 32-bit microcontroller-level RISC-V core with performance equivalent to that of an ARM M4. Like the Ariane platform, Pulpissimo's developers only support Xilinx based boards albeit more than one, though the PYNQ Z1 board is not one of them. It supports multiple interfaces such as SPI, I2C and UART, and has support for an instruction buffer. The TU custom platform on the other hand does not use any proprietary modules from any vendor and currently does support the PYNQ Z1 board, amongst others. It does not support I2C or SPI interfaces, but it does have access to a UART and can write to on-board LED's. Furthermore, it does come with configurable instruction and data caches and is being actively worked on by the department.

With identical cores and consequently similar performance, the selection was based on the required interfaces and support. As the proof of concept of the EMS module will be aimed at securing the RAM of a device, the fact that the custom platform supports instruction and data caches was a decisive factor. Without these, their influence on the performance costs cannot be measured or simulated. From this it followed that the TU custom processor was selected to be the development platform.

5.1.3. Specifications

The development platform is centred around a RISC-V RI5CY core [86]. This is a small 32-bit in-order microcontroller-level core with a 4-stage pipeline. It has full support for the $RV32I$ base integer, $RV32C$ compressed instructions and $RV32M$ integer multiply and division instruction sets extensions, as well as optional support for the $RV32F$ single precision floating point extensions [86] [87]. A block diagram of the core, taken from its user manual, can be seen in Figure 5.2. In this platform, its instruction interface connects only to the instruction cache through a modified $AHB-Lite$ bus. The data interface connects to another $AHB-Lite$ bus, which has access to the data cache as well as external peripherals including a UART, timer and board LED's. No flash interface is present. A rough overview of the setup is shown in Figure 5.3.

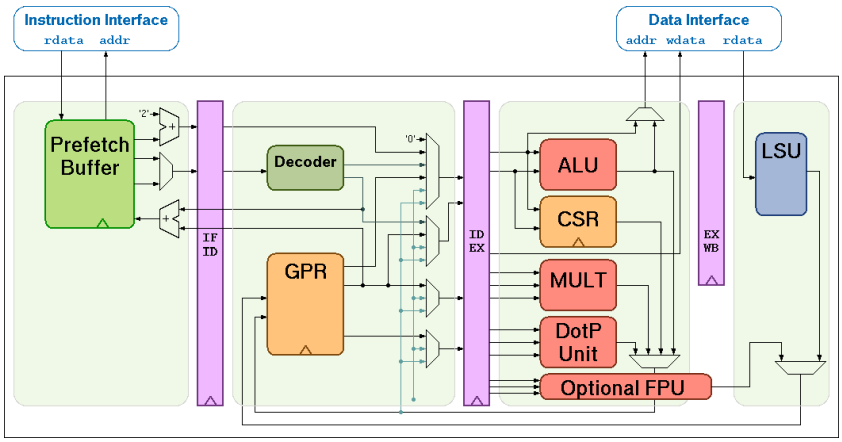


Figure 5.2: Block diagram of the RI5CY core [87]

The instruction and data caches of the platform are configurable. Their associativity, line size and line count can be set by changing some initialization parameters. During development and simulations they were set to 4-way set associative with a 128-bit line size, similar to other processors of its class [83], whereas its total size was varied through its line count. When synthesizing the platform to run it in hardware, caches larger than 2kb would no longer fit in the available FPGA. The caches respond in one cycle, meaning they either provide the data in case of a cache-hit, or forward the request to memory with one cycle of latency. They implement a round-robin replacement policy for simplicity, where the line that gets replaced during a miss depends on a modular counter. Finally, they are write-back caches, meaning updated lines are only written back to memory when that line is about to be replaced at a cache miss.

Upon a cache miss, data will be requested from memory. For this platform, the memory comes in the form of blockRAM on the FPGA as the development board does not support access to its external DDR3 interface directly from the FPGA. Since the platform does not have the hardware required to handle memory requests from both caches at the same time, each cache has its own separate range

of memory with its own interfaces. Each memory address contains one cache line worth of data, 128-bits, and read and write accesses to memory take a constant 100 cycles to complete, as is typical for external memory modules [71]. The EMS module is inserted between the cache and its memory to intercept and secure all data being transferred. This location is also shown in Figure 5.3.

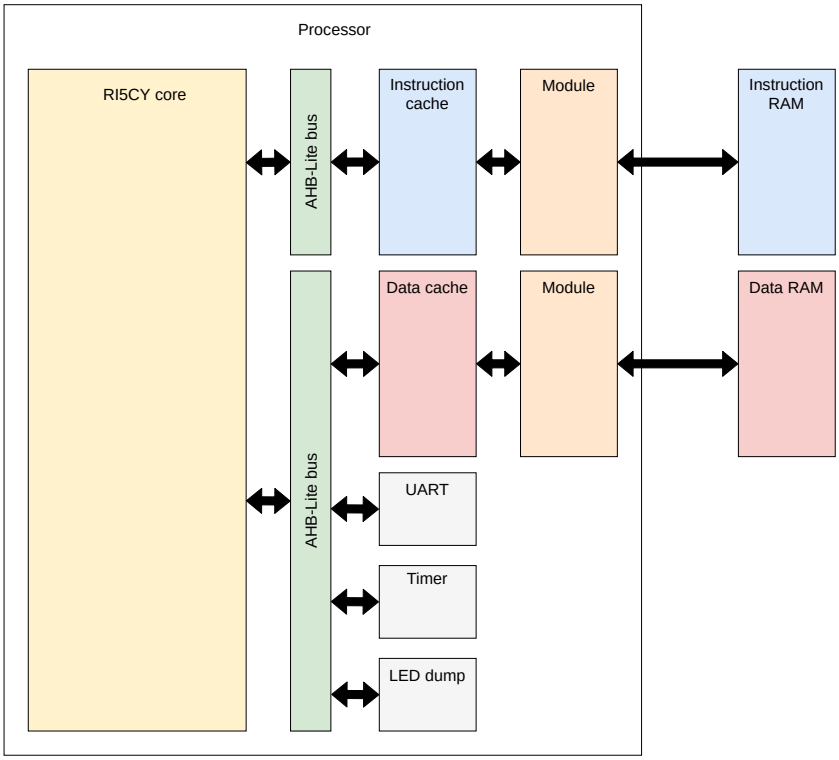


Figure 5.3: Schematic overview of the development platform, including location of EMS modules

Vivado 2019.2 is used on a Linux laptop to develop, test and integrate the EMS module into the platform. Changes are verified using the built-in simulator and when successful the design is synthesized and ran on the actual FPGA board. Benchmarks are written in C and compiled using the $riscv32 - elf - gcc$ toolchain, version 9.3.0 on the same laptop. After compiling, the resulting $.elf$ files are converted to $.dat$ to be loaded directly into the RAM upon initialization using some python scripts. These same scripts can be used to generate secured versions corresponding to the various EMS module variant covered later.

5.2. MAC

The integrity and authenticity verification part of the EMS module is performed with Message Authentication Codes (MAC). These use a secret key to generate a hash value for a given message. Upon reading the data, this value can be compared and verified. Without knowing the key, an attacker will not be able to generate any valid tags for custom data, ensuring authenticity. Furthermore, any changes in the data will lead to a different tag, ensuring its integrity.

For this purpose, a suitable hash function must be chosen. This section will first present the selection requirements. Next, a trade-off is provided between the available functions. Finally, the chosen hash function, SipHash, will be covered in more detail.

5.2.1. Requirements

For the variants of the EMS modules that support it, all data being stored to and read from memory must be verified. To do this, each block of data passing through the module is given a tag that is stored with it and verified when read back. Unlike hashes that are used to verify the integrity of a large file and need to be computed once, this application requires the verification of both integrity and authenticity of many small blocks of data.

Aside from the typical requirements of a MAC, such as that it should be infeasible to generate a valid hash without knowing the key or find another message that leads to the same hash value, there are some application specific requirements. These requirements greatly influence the selection of the hash function, and are as follows:

- **Req 1:** The first requirement is that the tag must be no larger than the data it is securing and will fit in a single memory line, as larger tags would take up a prohibitive amount of space. This platform stores 128-bits of data per memory address and as such that will be the upper limit of the hash output. The tag may be smaller, though if so, 64-bits is preferred for optimization reasons.
- **Req 2:** The second requirement is that calculating and verifying the MAC should be fast. As with encryption, any processing delays here will be added directly to the memory latency. Therefore, the time spent on this should be relatively short when compared to the overall memory delay.
- **Req 3:** The third requirement is that the MAC offers sufficient security. Similar to encryption, this is a complicated subject depending on many variables. In the context of this application, it simply means that there must be no known attacks that significantly reduce its security.
- **Req 4:** The fourth requirement is that the area requirements of the additional hardware are small, relative to its alternatives. Though perhaps a vague requirement, it means that hash functions designed to be lightweight are preferred.

5.2.2. Options

There are several ways to generate a Message Authentication Code (MAC). CMAC, for example, uses block-ciphers in Cipher Block Chaining (CBC) mode as shown in Figure 5.4. Similarly, cryptographic hash functions may be used along with a secret key, as is the case for HMAC. Aside from such constructions that make use of existing cryptographic primitives, there are some hash functions that are inherently designed to be used with a cryptographic key, making them MACs by definition.

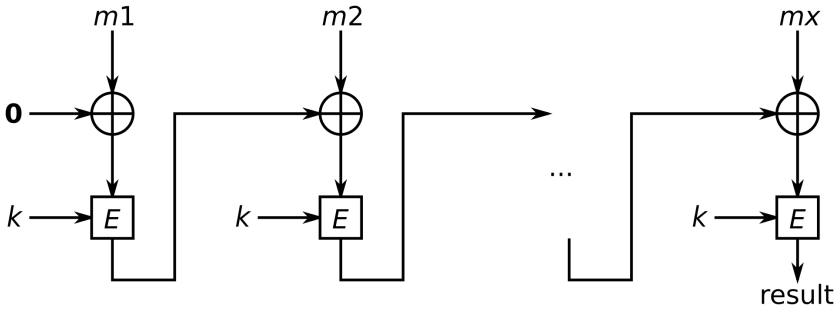


Figure 5.4: Overview of Cipher Block Chaining mode of operation to generate a MAC [88]

It was found that the first and second requirements removed most of the typical MAC solutions from the selection pool. Keccak for example, winner of the NIST SHA-3 competition, is a well-researched cryptographic hash function. Though faster than its predecessors used in SHA-1 and SHA-2, its hardware implementation is prohibitively large for our application, at tens of thousands Gate Equivalents (GE) [89]. Area optimized variants on the other hand exist but take several hundreds of cycles to complete, causing an unacceptable delay, while still taking up over 6500 GE of chip area. Furthermore, the short message size of 128-bits in our use-case would lead to inefficient application of many existing solutions that take time to initialize. DM-Present for example would have its throughput reduced by a factor of three or more compared to long messages [90]. As such, the search was focussed on lightweight MAC implementations optimized for small messages.

Where new lightweight block-ciphers were introduced constantly in recent years, the same can not be said for cryptographic hash and MAC functions. Biryukov et al. compiled an extensive collection of the state of the art in lightweight cryptography [91]. Their list includes over a hundred symmetric algorithms, most of which were block-ciphers, ten were hash functions and only two being dedicated MACs. Of the listed hash functions only four support inputs and outputs of 128-bits or smaller, namely; Armadillo, PHOTON, Spongent and GLUON. These, together with the two dedicated MAC functions SipHash and Chaskey made up the full selection list, and are briefly described next:

- Armadillo [92] was presented in 2010 as a general-purpose cryptographic function. It is aimed

to be implemented in hardware and to be used as a hash or MAC function, as well as a stream cipher in RFID tags.

- PHOTON [90] was presented in 2011, as a hash function aimed at hardware implementations in extremely constrained devices such as RFID tags. In particular, it is designed to require very little chip area and to be efficient with short messages.
- Spongent [93] was also presented in 2011. Similar to Armadillo and PHOTON it, too, is aimed at hardware implementations in RFID tags. Its permutation rounds are based on the Present [94] block-cipher. Of the considered options, Spongent has the lowest area requirements.
- GLUON [95] was presented in 2012, also aimed at RFID tags and embedded sensor networks. Its authors mostly compare it with PHOTON and state that it performs worse in most metrics, though think it is still relevant because of its well-studied construction.
- SipHash [96] was also presented in 2012. It is a dedicated MAC function optimized for short inputs, as well as to be fast in software and small in hardware. Originally intended to protect servers against hash-flooding attacks, it is now used in multiple other applications as well.
- Chaskey [97] is a MAC algorithm presented in 2014. Its main goal is to be fast when implemented in software and running on a microcontroller, when compared to AES-128-CMAC.

The specifications of all aforementioned options, as well as the to the best of my knowledge smallest Keccak implementation, are shown in Table 5.2. The area and latency per block are given for the 'typical' implementation as mentioned in the respective papers. Some also mention area-optimized implementations at the cost of greatly increased latency, and vice-versa. As both latency and area are of concern in our application, these non-balanced implementations are not considered. Similarly, as the sizes and latency of variants that produce tags greater than 128-bits are irrelevant to our application, these are omitted as well.

Table 5.2: Specifications of several lightweight hash and MAC functions. Data given for tag sizes smaller than 128-bit.

Name	Tag size (bits)	Block size (bits)	Area (GE)	Cycles per block
Keccak [89]	256	1600	>6500	6750
Armadillo [92]	80/128	48/64	4030/6025	44/64
PHOTON [90]	80/128	16	1168/1708	132/156
Spongent [93]	88/128	8	1127/1687	45/70
GLUON [95]	128	8	2071	66
SipHash [96]	64	64	3700	12
Chaskey [97]	128	128	N/A	N/A

From the table it becomes clear that PHOTON and Spongent have by far the smallest implementations, largely due to their construction and small block size. GLUON and SipHash both support 64-bit tags, of which two would fit in a single memory line. The four hash functions; Armadillo, PHOTON, Spongent and GLUON, would need additional hardware to let them function as a MAC. This would both increase their area and latency to some extent. SipHash is clearly the fastest, requiring only 12 cycles to process a 64-bit block. It is larger than PHOTON, Spongent and GLUON's implementations, but as it is already a MAC function by design, it requires no additional hardware for this functionality. For Chaskey, no physical implementation data could be found, as it is mostly aimed to be efficient in software applications on microcontrollers. Based on the given requirements and specifications stated above, SipHash was chosen to be the MAC function in the EMS modules.

5.2.3. Siphash

Siphash is a lightweight MAC function optimized for small input messages [96]. It was designed to counter hash-flooding attacks on servers, by preventing attackers from generating collisions. As such its main security goal is to prevent an attacker from guessing valid tags for any messages he has not seen before, even after having seen many other valid message/tag pairs. Messages are processed in 64-bit blocks using a 128-bit key and the results is a 64-bit tag. SipHash is used in several software applications for hash-table implementations, including Python, Bitcoin and systemd. Its operation is briefly described next.

Structurally, SipHash is rather straightforward. The authors define a single function RipRound

shown in Figure 5.5, as well as family of functions SipHash-c-d, where c represents the number of SipRound calls when compressing each block and d represents the amount of rounds during finalization. SipHash-2-4 is defined as the typical implementation, while SipHash-4-8 is proposed for higher security at the cost of speed. A run is initialized by XORing the key with four initialization vectors. Next, the message is processed 64-bits at a time, the final block being padded with zeros and a byte representing the length of the message. A full run of SipHash-2-4 can be seen in Figure 5.6.

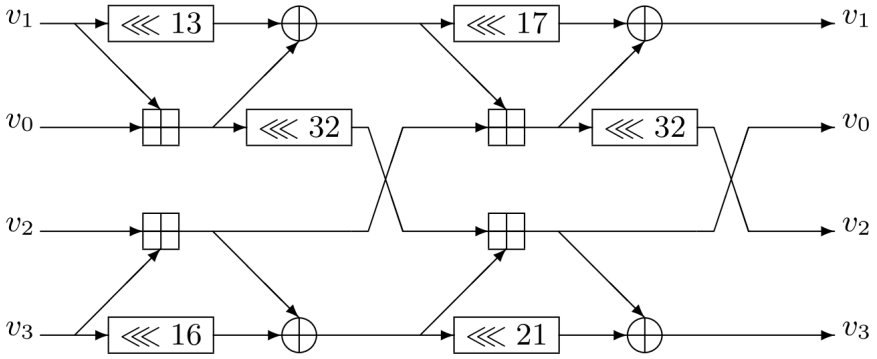


Figure 5.5: Overview of a single SipHash round [96]

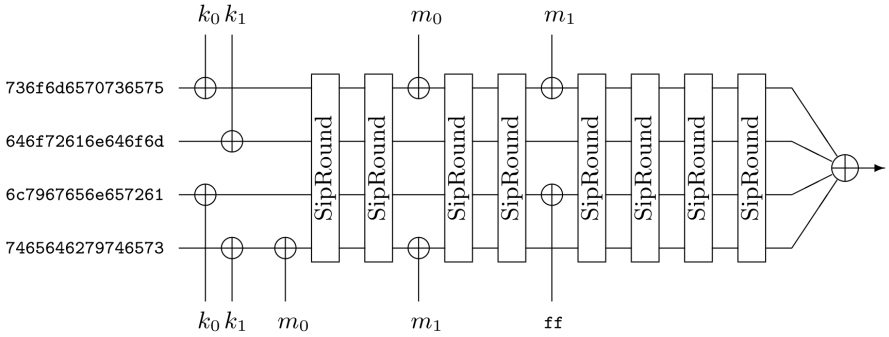


Figure 5.6: Overview of SipHash-2-4 generating a tag for a 15-byte message [96]

For a more complete description of the process as well as the reasoning behind the steps, the reader is referred to SipHash's research paper [96].

5.3. Cipher

To achieve the encryption task of the EMS module, a suitable cipher will need to be chosen. Since there is a wide variety of ciphers available, this section will first lay down the selection criteria relevant to our application. Next, a trade-off will be given between different options. Finally the workings of the chosen cipher, Prince, will be covered in more detail.

5.3.1. Requirements

With the EMS module variants that include it, all data passing through them must be encrypted. This means that for each transfer a small, fixed-size block of data needs to be processed as fast as possible. Such a setting is rather different from many typical cryptographic applications, and as such it puts some specific requirements on the chosen cipher. These main requirements are as follows:

- **Req 1:** The first requirement is that both encryption and decryption should be fast, since the time spent on processing data will be directly added to the delay caused by memory latency. This will linearly increase the amount of time the processor will have to wait during each memory access. The time, measured in clock cycles, spent on encryption must therefore be relatively short when compared to the overall memory latency.

- **Req 2:** The second criteria is that the required chip area for both the encryption and decryption functions should be small. The EMS module as a solution is meant to be applicable for devices ranging from low power nodes where size is a real concern, to larger processors where it might be less of an issue. The chosen development platform contains a microcontroller-level core, and as such a small cipher would be a realistic choice.
- **Req 3:** The third requirement is that the cipher is secure. This is a complicated subject when it comes to cryptography depending on many variables. Considering the application, this means it should uses a $128 - bit$ key similar to the chosen MAC function, and that there are no known attacks that significantly reduce the cipher's security.

5.3.2. Options

The typical first choice for encryption when security is a concern, is to use AES [98]. AES is used extensively in many applications and is regarded as the standard in cryptography. During its development in the late $1990's$ however, having a lightweight hardware implementation was not one of its design criteria. As such, the area and power requirement of such implementations are too large for embedded applications such as IoT devices. Since then there has been quite some development concerning lightweight cryptography, with many new ciphers being available.

When performing a search to find a suitable cipher, it was quickly found that the list is rather extensive and will need to be shortened significantly. As a first step, the options were limited to block ciphers only, as those fit best for the intended application: Each memory address stores a line of 128 bits of data, which is therefore the minimum amount of data to encrypt in one go. Furthermore, each of these addresses can be accessed at random and there is no guarantee that successive addresses will always be accessed in the same order. From this it follows that 128 bits is also the largest guaranteed amount of data to encrypt in one go. Though stream ciphers such as Grain [99] and Trivium [100] can be implemented very efficiently in hardware, they take several cycles for initialization followed by multiple cycles to encrypt a singly byte. This could be acceptable for situations where there is a continuous or unknown-length stream of data, but it makes them unsuitable for this application with short, constant blocks [101].

Next, the search was focused on ciphers with a fast and efficient hardware implementation, while retaining security. Several papers were found comparing large amounts of lightweight block ciphers based on area, power consumption, throughput and latency. In 2017 Hatzivasilis et al. started out by splitting the existing ciphers in three time periods [102]. They note that the first period started in the 1980's when cryptography was first introduced to mainstream computers. The second period ranges from 2005 to 2012 where embedded systems become prominent, triggering a race to develop ciphers with hardware requirements as small as possible. The third period includes the present where the focus shifts from area reduction to speed and latency improvements, as well as the possibility to decrypt with similar requirements. They then proceed to list several ciphers developed in each time period, and order them based on their construction type. This way they cover 56 block ciphers and 360 implementations, and compare them in multiple fields. Similarly, Banik et al. compare multiple hardware implementations of lightweight block ciphers based on their size and energy consumption [102]. They then explore the effects that unrolling rounds of each ciphers has on these properties, and create a model to predict the additional energy consumed for each additional unrolled round.

Comparing the different ciphers in a fair way was complicated. The comparisons by Hatzivazilis et al. are mostly focused on the encryption functionality and don't take into account the need for decryption as well, as is the case in our application. Furthermore, not all ciphers have been implemented in the same technologies or with the same key sizes, and some ciphers have been implemented multiple times in the same technologies with different performance figures. To make matters worse, some were implemented with varying amounts of unrolled rounds which also affects the area and throughput figures. Figure 5.7 for example shows the comparison results of several block ciphers, for several production technologies and key sizes [102]. Though not all ciphers were implemented in all technologies and the smaller bars are hard to read from the large area comparison (In Gate Equivalents (GE)), the authors also provide the raw data in a table. Banik et al. also provide their data in a table, but they have the same issue when it comes to consistent settings.

Through the requirements mentioned above and the ciphers covered in the comparison papers, the list of potential candidates was reduced to five entries:

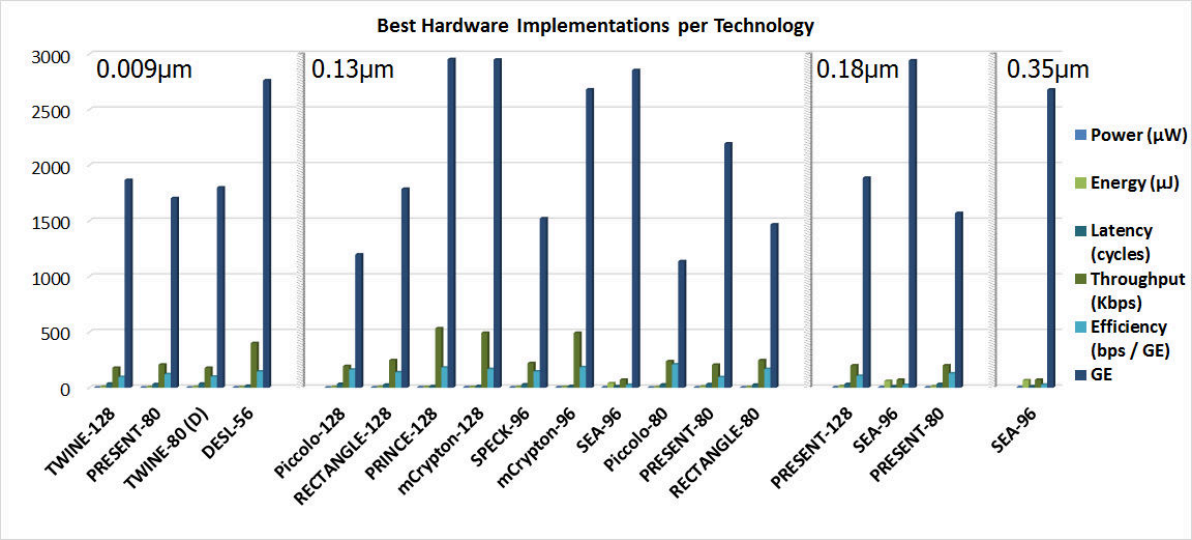


Figure 5.7: Comparison of several ciphers based on different criteria [102]

- mCrypton [103] was presented in 2005 as a more lightweight variant of the Crypton cipher, based around a Substitution Permutation Network (SPN). It uses 64, 96 or 128-bit keys to encrypt 64-bit data blocks in 25 rounds.
- Present [94] was presented in 2007 as one of the first block-ciphers specifically targeted at lightweight cryptography, and is currently the standardized block cipher for this application. It is built around an SPN, takes 31 rounds to process a 64-bit block of data and uses 80 or 128-bit keys.
- Piccolo [104] was presented in 2011 and is built around a generalized Feistel network aimed to be small and with low energy consumption. Adding decryption functionality is cheap compared to other lightweight ciphers. It processes 64-bit blocks of data in 25 and 31 rounds for the 80 and 128-bit key variants respectively.
- Prince [105] was presented in 2012, specifically designed to be lightweight while providing high throughput and low latency. It is a special type of SPN based cipher that takes 12 rounds to process 64-bit data blocks using a 128-bit key. Unlike the other ciphers, it was designed for all rounds to be fully unrolled to process a block in one clock cycle. Due to its construction no additional hardware is required for the decryption functionality.
- Rectangle [106] was presented relatively recently in 2015. It is also based around a SPN structure, takes 64-bit blocks to process in 25 rounds and supports keys of 80 or 128-bits. It is aimed specifically at devices such as RFID tags, sensor nodes and smart cards.

In an attempt to fairly compare these options, the best implementations of each cipher mentioned in [102] for a given technology are shown in Table 5.3, where possible with the same key size and including decryption functionality marked with (D). AES is included as well to indicate the reduction in area requirements for lightweight ciphers. From this data it is clear that Prince is the most efficient when it comes to raw throughput per area of the cipher, followed by Piccolo and Rectangle. Decryption functionality is only included for the Piccolo and mCrypton implementations however, and for Present only the 80-bit key variant is given. Though Prince uses the same hardware for both encryption and decryption and adding this functionality won't increase its area by much, this is not necessarily the case for the other ciphers. Furthermore, the authors of Prince state that it was designed to be efficiently fully unrolled, in order to process a block in a single clock cycle. The mentioned implementation on the other hand is iterative and takes 12 cycles.

The comparisons in [107] are similarly limited, but do shed some light on the effects of unrolling rounds of the ciphers. Their data for the relevant ciphers is shown in Table 5.4. The throughput and efficiency columns were not part of the original data, but were calculated using the block size, amount of cycles and the cipher's area in order to give a figure that can be compared with the data in Table 5.3. It should be noted that both Piccolo and Present are implemented with an 80-bit key, which makes the

Table 5.3: Hardware performance of block ciphers on 0.13μm technology. Truncated data taken from Table 5 in [102]

Cipher	Key size (bits)	Block size (bits)	Cycles per block	Throughput at 100kHz (kbps)	Area (GE)	Efficiency (kbps/KGE)
AES	128	128	226	48	11031	4.35
mCrypton (D)	128	64	13	492.3	4108	119.83
Present	80	64	31	206	2195	93.84
Piccolo (D)	128	64	33	193.8	1362	142.32
Prince	128	64	12	533.3	2953	180.59
Rectangle	128	64	26	246	1787	137.66

implementation both smaller and faster than their 128 counterparts. The AES implementation shown here also takes considerably less cycles to process a block compared to the previous data, at the cost of additional hardware space. No data is present for mCrypton and Rectangle, and no information is given about any of the implementations' support for decryption functionality. Nonetheless, Prince is again the most efficient when it comes to throughput over area. In fact, when Prince is fully unrolled to process a full block in one cycle as per its original design, it is less than twice the size of the most unrolled 80-bit Piccolo variant which still takes 8 cycles, while consuming only half as much energy per bit.

Table 5.4: Estimated hardware performance of block ciphers on STM 90nm low leakage technology @ 10MHz. Truncated data taken from Table 4 in [107]

Cipher	Key size (bits)	Block size (bits)	Unrolled rounds	Cycles per block	Calculated throughput at 100kHz (kbps)	Area (GE)	Calculated efficiency (kbps/KGE)	Energy per bit (pJ)
AES	128	128	1	11	1163	12459.0	93.4	2.74
			2	6	2133	22842.3	93.3	4.64
			3	5	2560	32731.9	78.2	8.15
			4	4	3200	43641.1	73.3	11.07
			5	3	4266	53998.7	79.0	12.77
Present	80	64	1	33	193	1439.9	134.6	2.69
			2	17	376	1967.9	191.3	2.43
			3	12	533	2499.3	213.4	2.79
			4	9	711	3000.4	237.0	3.13
Piccolo	80	64	1	26	246	1492.0	164.9	2.78
			2	14	457	2385.5	191.6	4.42
			3	10	640	3268.1	195.8	6.55
			4	8	800	4124.7	193.9	9.45
Prince	128	64	1	13	492	2286.5	215.4	2.33
			Half	3	2133	8245.9	258.7	5.60
			Full	1	6400	7728.6	828.1	5.77

Based on the mentioned requirements, namely that encryption should be fast and small, it is clear that Prince is the cipher of choice. To the best of my knowledge there are no known attacks on the Prince cipher that significantly reduce its security level, aside from some related-key attacks that are not relevant to this setting [108]. The workings of Prince will be briefly covered next.

5.3.3. Prince

Prince is a lightweight block cipher optimised to provide low latency encryption and decryption of data, proposed in 2012 by Borghoff et el. [105]. It was designed to be fully unrolled in hardware, processing a 64-bit block of data within one single clock cycle with a 128-bit key, while maintaining a short critical path to keep the clock rates 'moderately high'. Designing it this way provided some new possibilities by removing the need for very similar round functions, as is the case for iterative hardware implementations. Furthermore, it allows for a design where implementing decryption comes at a minimal additional cost. With Prince, decrypting with one key is the same operation as encrypting with a related key, a

property the authors refer to as 'α-reflection'. As such no, additional hardware is required for decryption, aside from a basic operation on the secret key.

Structurally, Prince is a type of Substitution Permutation Network (SPN) similar to AES and many other block ciphers. As their name implies, these networks perform several rounds consisting of substitution operations using a so-called S-boxes, and permutation operations using P-boxes to generate a ciphertext from a given plaintext and key. The goal of the S-box is to replace a block of bits with another block of bits, where changing one bit of the input block leads to ideally half the bits of the output block being changed. This property is referred to as an avalanche effect and ensures that similar plaintexts do not result in similar ciphertexts and vice-versa. To save code or hardware space, S-boxes generally only operate on parts of the input and multiple are used in parallel to process the full width of the block. The goal of the P-box is to spread the outputs of the S-boxes to the inputs of as many S-boxes in the next round as possible by shuffling the bits. By performing multiple rounds of substitution and permutation steps, this ensures that all bits in the full ciphertext depend on each bit of the plaintext. Decrypting is done by inverting the S- and P-boxes and reversing their order in a round.

The encryption and decryption process of Prince consists of five distinct steps: **(1)** a key whitening step; **(2)** five regular rounds; **(3)** a middle round; **(4)** five inverted round; and **(5)** a final key whitening step. The outline of these steps is shown in Figure 5.8 and Figure 5.9. The keys k_0, k'_0 and k_1 are 64-bit subkeys generated from the 128-bit master key k as follows:

$$k = k_0||k_1$$
$$(k_0||k_1) \rightarrow (k_0||k'_0||k_1) := (k_0||(k_0 >>> 1) XOR (k_0 >> 63)||k_1)$$

Figure 5.8: Top level overview of Prince block cipher [105]

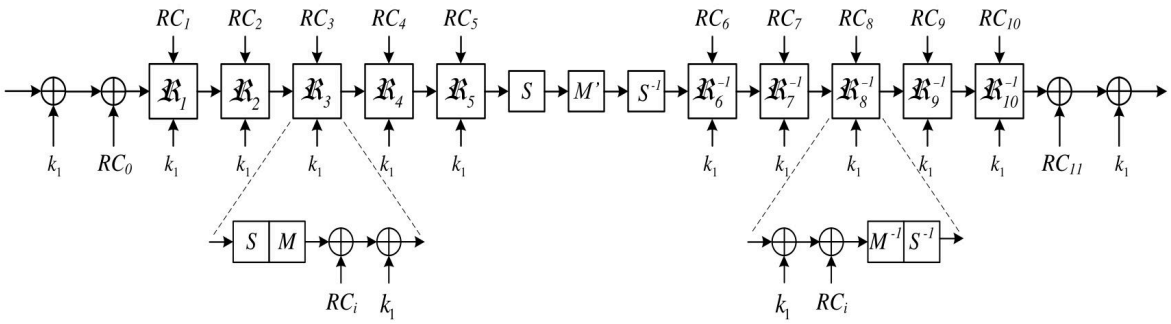


Figure 5.9: Operational structure of the Prince core and its rounds [105]

The steps performed by the Prince core are explained next:

1. $\mathbf{k_1}$ **add** XOR's the 64-bit state with the constant 64-bit sub-key k_1. Prince uses the same sub-key for each round.
2. $\mathbf{RC}_i$ **add** XOR's the 64-bit state with a given 64-bit round constant shown in Table 5.5.

Table 5.5: Round constants used by the the Prince block cipher

RC_0	0000000000000000
RC_1	$13198a2e03707344$
RC_2	$a4093822299f31d0$
RC_3	$082efa98ec4e6c89$
RC_4	$452821e638d01377$
RC_5	$be5466cf34e90c6c$
RC_6	$7ef84f78fd955cb1$
RC_7	$85840851f1ac43aa$
RC_8	$c882d32f25323c54$
RC_9	$64a51195e0e3610d$
RC_{10}	$d3b5a399ca0c2399$
RC_{11}	$c0ac29b7c97c50dd$

3. **S-layer** substitutes each 4-bit nibble x of the 64-bit state according to the 1D S-box shown in Table 5.6.

Table 5.6: S-box used by the the Prince block cipher

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	B	F	3	2	A	C	9	1	6	7	8	0	E	5	D	4

4. **M/M'-layers** perform the function of a P-box by diffusing the bits of the 64-bit state. M' is a $64x64$ matrix as defined below with which the 64-bit state is multiplied. This matrix is an involution, meaning it is its own inverse. The M and M^{-1} steps consist of a multiplication with M', followed by an additional shift-rows permutation of the state shown in Table 5.7, or its inverse, respectively.

$$M_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, M_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} M_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, M_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\hat{M}_0 = \begin{pmatrix} M_0 & M_1 & M_2 & M_3 \\ M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \end{pmatrix}, \hat{M}_1 = \begin{pmatrix} M_1 & M_2 & M_3 & M_0 \\ M_2 & M_3 & M_0 & M_1 \\ M_3 & M_0 & M_1 & M_2 \\ M_0 & M_1 & M_2 & M_3 \end{pmatrix}$$

$$M' = \begin{pmatrix} \hat{M}_0 & 0 & 0 & 0 \\ 0 & \hat{M}_1 & 0 & 0 \\ 0 & 0 & \hat{M}_1 & 0 \\ 0 & 0 & 0 & \hat{M}_0 \end{pmatrix}$$

Table 5.7: Shift-rows permutation used by the the Prince block cipher during M step

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SR[x]$	0	5	A	F	4	9	E	3	8	D	2	7	C	1	6	B

This symmetric structure is what allows the so-called α-reflection property to occur. Because of this property it is possible to decrypt messages with the same hardware used to encrypt them, saving significantly on area requirements. For a full explanation on how this works exactly as well as the reasoning behind the design of the rounds, the reader is referred to Prince's research paper [105].

5.4. EMS Modules

With the platform, cipher and MAC function now set, several variants of the Embedded Memory Security module were implemented as described in Chapter 4. Aside from the three types mentioned there that

perform encryption, integrity verification an both, two additional variants were developed that optimize memory usage for the MACs. This resulted in the five different types described as follows:

- **Encrypt:** The most straightforward implementation only ensures the confidentiality of data by encrypting any transfers to and from memory with the Prince cipher.
- **MAC (Single):** This variant verifies the integrity and authenticity of transfers to and from memory. It takes a full 128-bit memory line to store its 64-bit MAC as calculated with SipHash for each line of data, with the remaining 64 bits padded with zeroes. As such, each memory access requires an additional access for the MAC and capacity requirements are doubled.
- **MAC (Double):** This is an optimized version of the MAC (Single) variant. It reduces memory requirements to 33.3%, by storing the 64-bit MAC's of two adjacent memory lines in one 128-bit line. Writing to memory now takes an additional read and write to update the MAC.
- **Encrypt & MAC (Single):** This variant ensures the confidentiality, integrity and authenticity of data in memory. It encrypts each transfer and verifies a MAC over the encrypted data. One MAC is stored per memory line.
- **Encrypt & MAC (Double):** This variant is the memory-optimized version of the Encrypt & MAC (Single) module, similar to above.

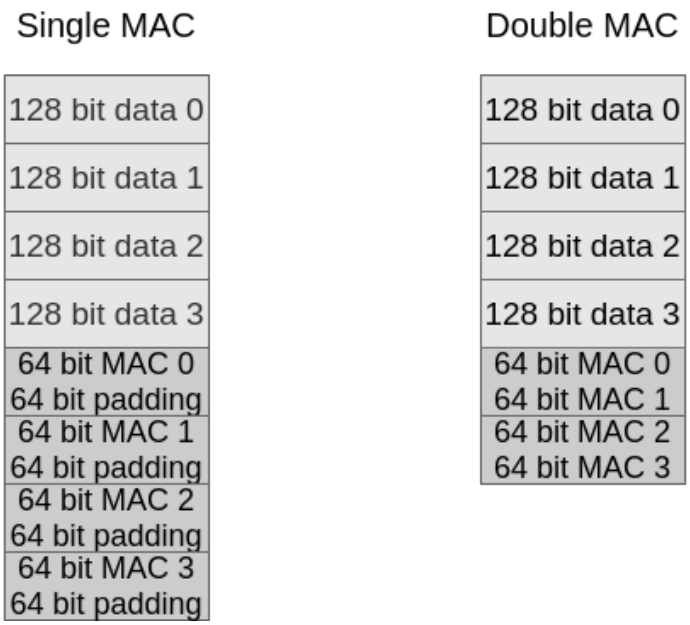


Figure 5.10: Schematic memory layout for Single and Double MAC variants of the EMS modules

The EMS modules store their MAC tags in a region in memory that is not accessible by the processor. In this platform, the RAM address bus coming from the processor is 11 bits wide. This way, the processor can index 2048 lines of memory, for a total of 32kb or RAM. The address bus between the EMS modules and the external memory is expanded to 12 bits, allowing to access a total of 64kb. For the single-MAC variant, the MAC address is simply the data address offset by 2048. The double-MAC variants store the MACs of two adjacent data addresses in the same location to save capacity, at the cost of an additional read to update the correct MAC when writing data to memory. This concept is shown schematically in Figure 5.10.

5.4.1. Platform Interfaces

The EMS modules are installed in the data-path between the processor and its memory controller. As such, they have matching interfaces to connect to both. The interfaces present in this processor platform consist of several signals, all of which are schematically shown in Figure 5.11 for an encrypt and MAC version, and listed as follows:

- **Request:** The request signal originates from the processor. It is a single bit signal that is pulled

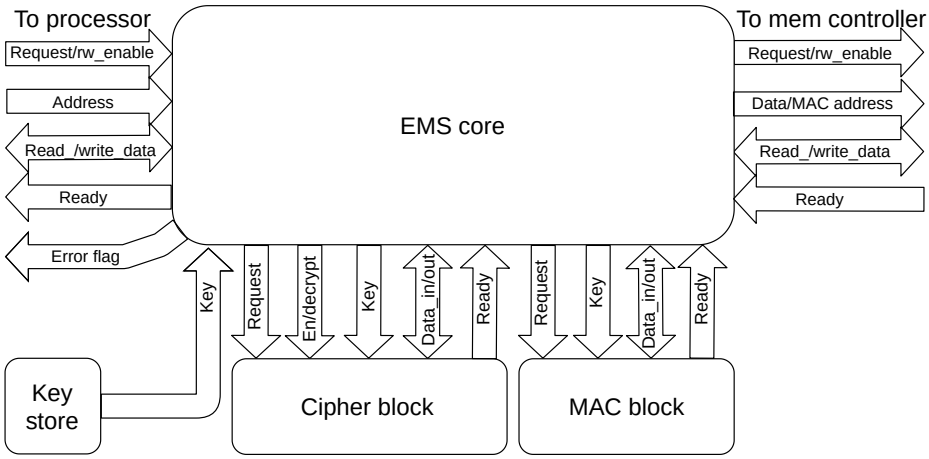


Figure 5.11: Interface overview of an encrypt + MAC EMS module

high when the processor intends to initiate a data transfer to or from memory. This signal is kept high for the duration of the request, until the memory signals ready.

- **Rw_enable:** The read/write enable signal originates from the processor. It is a single bit signal that is pulled high during a request to signify a write to memory, or pulled low to signify a read.
- **Address:** The address signal originates from the processor. It is a 11-bit signal and signifies the data address in memory to access. This way the processor can index 2048 memory lines, each containing 128 bits for a total of 32kb.
- **Write_data:** The write data signal originates from the processor. It is a 128-bit signal that transfers one memory address worth of data to the memory during a write request.
- **Read_data:** The read data signal originates from the memory. It is a 128-bit signal that transfers one memory address worth of data to the processor during a read request.
- **Ready:** The ready signal originates from the memory. It is a single bit signal that is raised for one clock-cycle once the memory has completed a request from the processor, and pulled low otherwise. During a write this indicates that data has been stored successfully and during a read this signifies the data on the read_data signal is valid.

Aside from these signals, the EMS modules that perform integrity verification include an additional connection towards the processor:

- **Error:** The error signal originates from the EMS module. It is a single bit signal that functions as a flag and is pulled high when a MAC miss-match is detected, signalling the processor that memory integrity has been compromised. The flag remains high until a device reset.

The functions of the other signals remain the same with the EMS modules present and, from the memory and processor's point of view, the modules are transparent. Only the address bus on the memory side is extended with one additional bit, not accessible by the processor. Instead, it is used by the EMS modules to index the addresses where the MACs are stored.

5.4.2. Components

Aside from the platform interfaces, the cores of the modules are also connected to their respective cryptographic hardware: The cipher and MAC blocks. The interfaces to these blocks as well as their structure and operation are briefly described next.

Cipher Block

The cipher block implements the Prince block-cipher for the encryption functionality of the EMS modules. It has the following connections with the EMS core.

- **Request:** The request signal originates from the EMS core. It is a single bit signal that is pulled high to initiate an encryption or decryption operation. This signal is kept high for the duration of the request, until the cipher block signals ready.

- **Encrypt/decrypt:** The encrypt/decrypt signal originates from the EMS core. It is a single bit signal that is pulled high during a request to signify encryption, or pulled low to signify decryption of the input data.
- **Key:** The key signal originates from the EMS core. It is a 128-bit bit signal that transfers the key with which to encrypt or decrypt the input data.
- **Data_in:** The data_in signal originates from the EMS core. It is a 128-bit signal that transfers one memory address worth of data to the cipher block, to be encrypted or decrypted.
- **Data_out:** The data_out signal originates from the cipher block. It is a 128-bit signal that transfers one memory address worth of data to the EMS core after encryption or decryption is complete.
- **Ready:** The ready signal originates from the cipher block. It is a single bit signal that is raised for one clock-cycle once the cipher block has completed the requested encryption or decryption of its input data. This indicates that the output data is valid.

At the centre of the cipher block is the prince core. The original design of this core was taken from [109]. It has been extended to include decryption functionality according to the cipher's specifications, as outlined in Section 5.3. This core can process a single 64-bit block of data in one clock cycle.

Upon receiving a request from the EMS core, the cipher block modifies the input key based on the encrypt/decrypt signal. The resulting key is then supplied to the Prince core, along with the first 64-bit half of the 128-bit input data. It then generates the first 64-bits of the output data. The second 64-bit block is processed the following cycle, completely encrypting or decrypting the input data in two clock cycles. The ready flag is then raised and the full 128-bit block of output data is presented to the EMS core.

MAC Block

The MAC block implements the SipHash MAC function for the integrity and authenticity verification functionality of the EMS modules. Its interfaces to the module's core are defined as follows:

- **Request:** The request signal originates from the EMS core. It is a single bit signal that is pulled high to initiate a MAC calculation. This signal is kept high for the duration of the request, until the MAC block signals ready.
- **Key:** The key signal originates from the EMS core. It is a 128-bit signal that transfers the key with which to calculate the MAC of the input data.
- **Data_in:** The data_in signal originates from the EMS core. It is a 128-bit signal that transfers one memory address or cache line worth of data to the MAC block, to be processed.
- **Data_out:** The data_out signal originates from the MAC block. It is a 64-bit signal that transfers the calculated MAC to the EMS core after processing is completed.
- **Ready:** The ready signal originates from the MAC block. It is a single bit signal that is raised for one clock-cycle once the MAC block has completed the calculation of the MAC over the input data. This indicates that the output data is valid.

At the centre of the MAC block is the SipHash-2-4 core. The original design of this core was taken from [110]. Its reset handling was slightly modified to operate the same as the rest of the platform. This core operates according to the specifications outlined in Section 5.2. It processes data in blocks of 64-bits at a time, performs two processing rounds during compression between each block and performs four rounds for finalization after the last block has been compressed.

Upon receiving a request from the EMS core, the MAC block initializes the SipHash core. The next cycle, it supplies the it with the first 64-bit half of the 128-bit input data, and starts a compression round. When the core is done, the next 64-bit block is provided. After this, in line with the specifications of SipHash, a third block specifying the length of the message is provided, padded with zeroes. Once all three blocks have been compressed, the core is instructed to finalize the MAC, after which the result is provided to the output of the MAC block to the EMS core. The process of calculating the MAC takes a total of 11 clock-cycles.

6

Results

After the Embedded Memory Security modules were developed as described in the previous chapter, they were tested for functionality and performance. This chapter will first describe the setup and performed experiments in Section 6.1. Next, the performance impact of the modules is measured and tested based on benchmarks in Section 6.2. Following this, the effects of cache-sizes on this performance is explored in Section 6.3. Section 6.4, then, presents study cases against some attacks, and Section 6.5 covers the hardware cost and overhead of the modules. The chapter is closed with Section 6.6 comparing the cost of the modules to some related works, and finally Section 6.7 providing a discussion of the results.

6.1. Setup

This section first provides an overview of the rest of this chapter, discussing the performed experiments and the tools used.

6.1.1. Performed Experiments

To determine the performance of the EMS modules from all aspects, five groups of experiments were performed. These experiments are as follows:

1. **Performance Evaluation:** First, the additional delay caused by the modules per memory access is measured. Software benchmarks are then ran on the development platform with the different module variants, keeping all other variables the same, to measure the actual in-use performance. These runs are performed with a 4kb, 4-way L1 cache.
2. **Cache Impact Evaluation:** To measure the effects of cache miss-rates on the performance impact of the modules, the same benchmarks are ran with caches of 2kb, 8kb and 16kb in size.
3. **Security Evaluation:** The added security of the modules is explored through three study cases. Here, attacks are performed against the platform to observe the effects of the various modules.
4. **Hardware Overhead Evaluation:** All variants of the EMS module were synthesized and implemented in an FPGA, along with the processor platform, to determine their area and timing requirements.

6.1.2. Benchmarks

Five benchmarks were selected to measure the performance impact of the different module variants, under various workloads. These benchmarks were taken from the riscv-tests repository [111] and ported to run on this platform. They were compiled using the $riscv32-elf-gcc$ toolchain, version $9.3.0$, which was locally generated. Various Python scripts were written to convert the resulting $.elf$ files into a format that could be loaded into the processor's memory by Vivado. These same scripts also encrypted and hashed the contents to initialize the runs for the corresponding modules. A short description of each benchmark is provided next:

1. The **Median** benchmark performs a basic three-element 1D median filter over a 400 element input

array. Afterwards, the result is verified by comparing with an included results array. If a mismatch is found, the benchmark returns with an error code.

2. The **Multiply** benchmark multiplies the elements of two 100 element input arrays one-on-one through a software shift-and-add algorithm. The resulting array is then again compared to an included array with correct results.
3. The **Qsort** benchmark implements the quicksort algorithm on a 2048 element input array. The elements are then sorted in ascending order and the results are verified. This benchmark has the largest dataset and as such the most memory operations of the five, but also performs a computationally intensive algorithm.
4. The **Towers** benchmark is a purely arithmetic intensive algorithm with no real dataset. It plays a round of the Towers of Hanoi puzzle. Here, a player must move rings of various sizes between three pegs in order to stack them from large at the bottom to small at the top. This benchmark performs the puzzle with 10 rings.
5. The **Vvadd** benchmark adds the elements of two 300 element input arrays one-on-one. It too compares its results with an included array. As such it is heavily data oriented with low computational intensity.

These benchmarks were chosen to represent various workloads from data-heavy such as Vvadd, to computationally intensive such as Towers. Simulating a single benchmark would take several minutes to complete on the used laptop, despite taking only milliseconds in hardware time. An example of the traces generated by a benchmark run is shown in Figure 6.1.

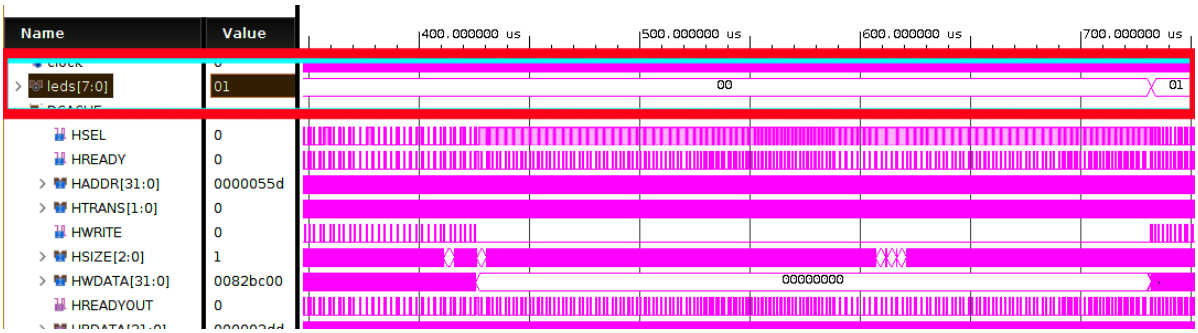


Figure 6.1: Example traces of a simulated benchmark run in Vivado, with inverted colours. Note the leds output changing value, indicating a run's completion.

6.1.3. Hardware Platform and Tools

The EMS modules were developed in the Verilog hardware description language, according to the specifications laid out in the previous chapter. To simulate and verify their functionality, Xilinx Vivado 2019.2 was used running in Arch Linux on a laptop machine. After integration into the development platform, the whole system was simulated to run benchmarks. The modules could only be used to protect the system's RAM, as the platform does not have any external flash devices. Similarly, since the platform does not support both the instruction and data caches to interface with the same RAM, two of the modules had to be added, one for each cache. A single address in RAM stores 128-bits of data, corresponding to one cache line, with a single memory access having a latency of 100 clock cycles. Benchmarks were ran with 4-way L1 caches of 2kb, 4kb, 8kb and 16kb in size, and no L2 or higher levels. Simulating caches of 32kb became prohibitively slow and as will be shown later, would not have had any noticeable further effect on performance.

The platform was also synthesized in Vivado and physically ran in hardware. This was done on the available PYNQ-Z1 board [81] described in Chapter 5 and shown again for clarity in Figure 6.2. As the FPGA could only contain the platform with a 16-way variant of the smallest $2kb$ caches, this was done mostly to verify that the modules work correctly in hardware and to measure their hardware overhead.

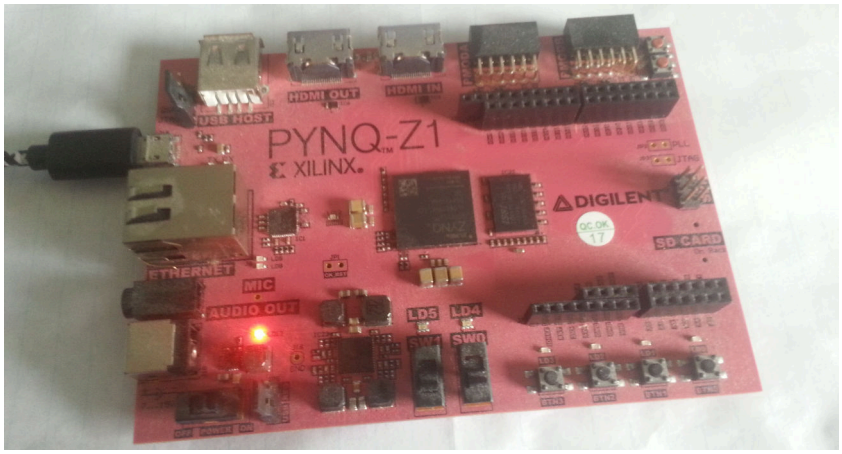


Figure 6.2: Picture of the available fpga development board, a Xilinx PYNQ-Z1. Repeat of Figure 5.1

6.1.4. Security Evaluation Scenarios

The developed modules are intended to protect a device against hardware-based attacks on its external memory. For completeness, the effects of manipulating the contents of memory are analysed for the different modules. Three attacks are selected and performed for this purpose, these are listed as follows:

1. **Fault/Code/Data injection:** The main concern is that an attacker is able to alter running applications or write his own modified software to the device. Countering this is the main goal of the EMS modules. Fault injections are similar, as data in memory is changed to a value of the attacker's choosing.
2. **Rogue memory:** Here, an attacker may try to change the full contents of the device's memory, or even swap the physical memory chip itself with another one. Then, the firmware or running state of applications could for example be changed to an earlier version, perhaps with known vulnerabilities, obtained from another device.
3. **Replay attacks**: These differ from the above injection and rogue memory attacks, in that the attacker changes contents of the same memory back to an earlier state. This implies that he has access to valid MACs for the intended manipulation. Although it would be complicated to implement custom applications this way, it would be possible to 'restore' the device to an earlier version of its firmware, or cause unintended behaviour.

6.2. Performance Evaluation

The additional delay that the Embedded Memory Security modules impose on the duration of a memory access was first determined. These results can be seen in Table 6.1. The longest additional time is caused during writes by the encryption and MAC (Double) variant of the modules, taking 37 cycles to perform its processing. The additional two memory accesses required to update the MAC far outweighs this processing time.

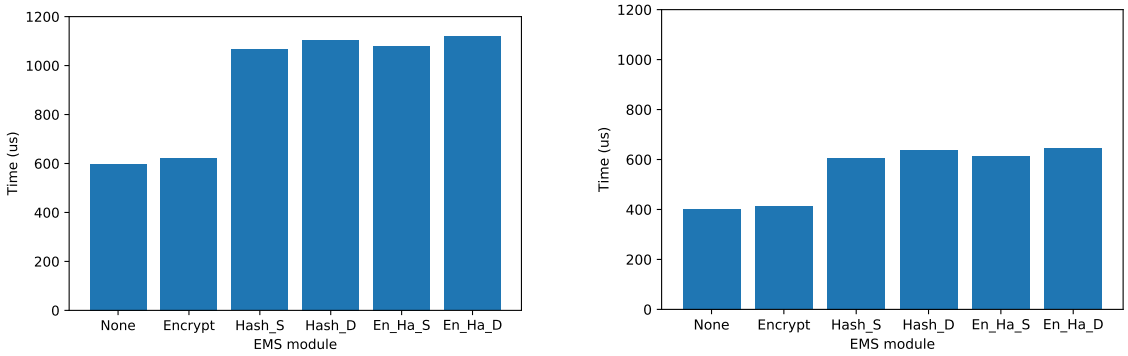
Table 6.1: Delays in clock cycles of a single cache request to read or write to memory, with and without the various EMS modules.

Module :	Read:	Write:
None	100	100
MAC (Single)	230	229
MAC (Double)	230	332
Encrypt	108	108
Enc + MAC (S)	235	234
Enc + MAC (D)	235	337

As the processor has caches, the somewhat pessimistic situation drawn by the table does not reflect

real-life performance. To more accurately determine this impact, the five benchmarks as selected above were ran on the platform. Since the benchmarks follow a set sequence of instructions and all other aspects of the platform remain the same, any differences in execution time are caused purely by the modules. This way these times can be directly compared to those of the baseline platform. For this experiment the platform's instruction and data caches are 4-way set associative and 4kb in size. Each benchmark was successively ran 10 times to measure both cold-start and average performance. Due to the data structure of the Qsort benchmark and the limited amount of memory available, it could only be ran once without requiring a complete re-write. Following runs would receive the already sorted array as input, leading to a different execution from the first run and therefore no comparable results. The results of each benchmark are discussed next.

6.2.1. Median



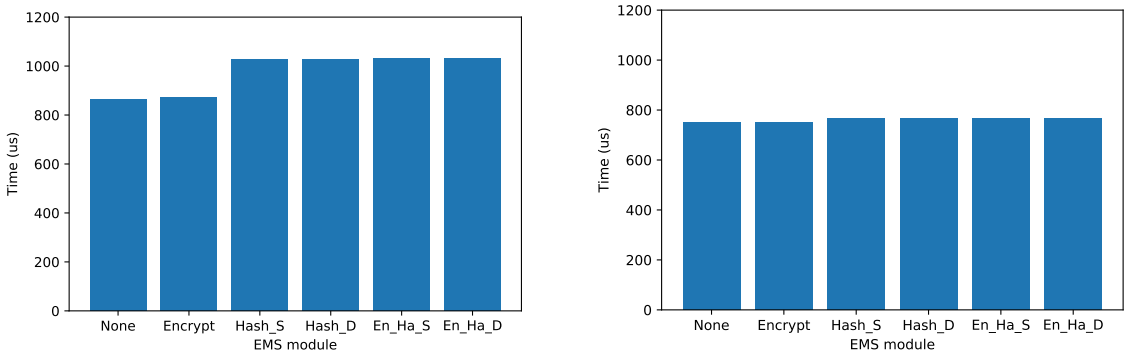
(a) Cache cold-start

(b) 10 runs averaga

Figure 6.3: Execution times of the Median benchmark

Figure 6.3 shows the execution times for the Median benchmark. The effects of the memory integrity verification are clearly visible in the cold-start runs. Execution times are nearly doubled, though this increase is reduced to 50% over the next ten runs. The memory-optimized variants of the EMS module show a slight increase over the single-hash variant which is caused by cache-misses after the cold start, in particular its additional writes to memory.

6.2.2. Multiply



(a) Cache cold-start

(b) 10 runs average

Figure 6.4: Execution times of the Multiply benchmark

The execution times for the Multiply benchmark are shown in Figure 6.4. For the cold-start runs it can be seen that there is a noticeable decrease in performance, caused by the EMS module variants that verify memory integrity. This effect almost completely disappears during subsequent runs when the

data is already in the cache and fewer memory accesses are required. The purely encryption version of the EMS module does not have a significant impact on performance, even for the cold-start runs.

6.2.3. Qsort

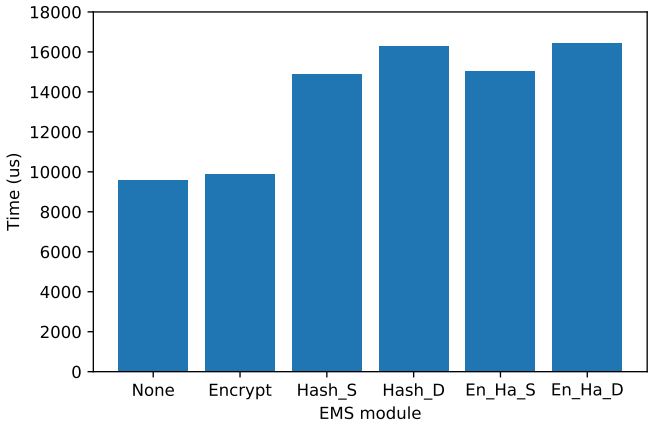
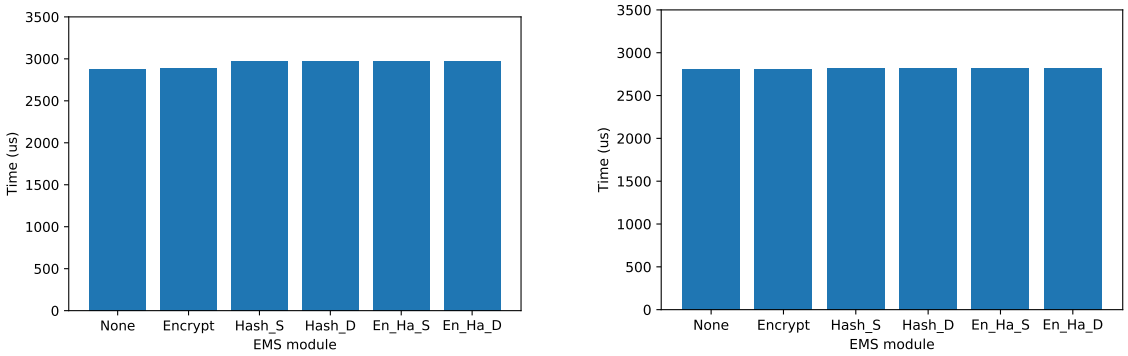


Figure 6.5: Execution times of the Qsort benchmark

The cold-start results for the Qsort benchmark can be seen in Figure 6.5. Here, the distinction between single and double-hash variants is most visible of all benchmarks. This shows that there is a significant amount of writes to memory, indicative of cache misses during execution after the initial cold-start. As mentioned above, due to the datastructure of the benchmark and the available memory, no subsequent runs were performed.

6.2.4. Towers

Figure 6.6 shows the impact of the EMS modules on the execution times of the Towers benchmark, or rather, the lack thereof. As mentioned above, Towers is a purely arithmetic intensive benchmark with no real dataset to process, leading to very little cache-misses. This results in few memory accesses where the modules would have an effect. The cold-start run times are therefore also very close to the averages.



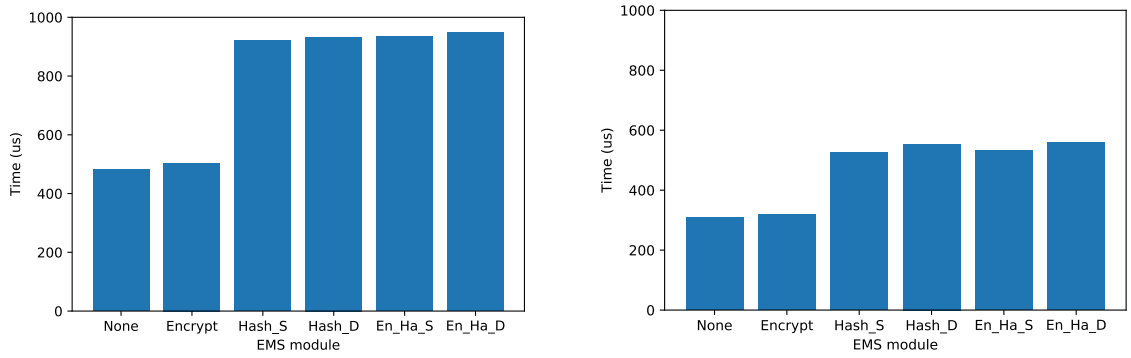
(a) Cache cold-start

(b) 10 runs Average

Figure 6.6: Execution time of Towers benchmark

6.2.5. Vvadd

Finally, Figure 6.7 displays the results of the Vvadd benchmark. It shows a process similar to that of the Median benchmark: The addition of the integrity verifying modules close to doubles the execution



(a) Cache cold-start

(b) 10 runs average

Figure 6.7: Execution time of Vvadd benchmark

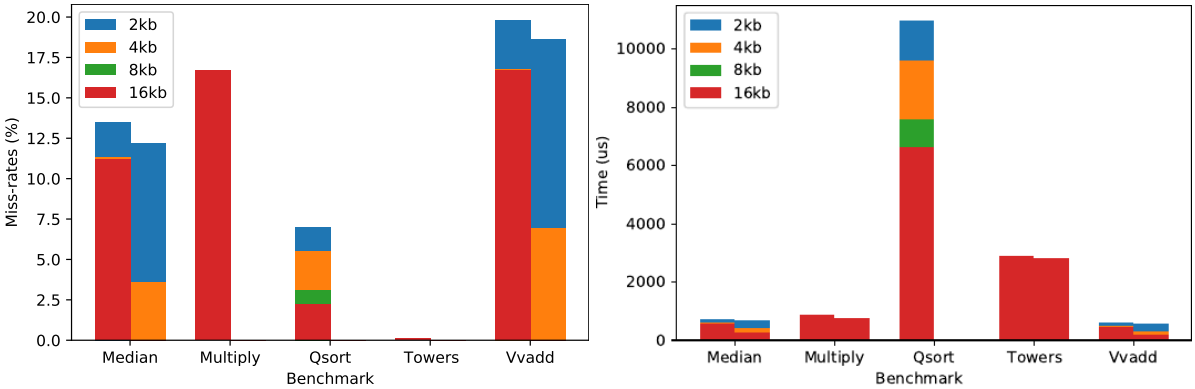
times of the cold-start runs. This effect becomes smaller over the next runs as data is already loaded in the caches.

6.3. Cache Impact Evaluation

This section explores the effects of different cache-sizes on the performance impact of the EMS modules. As the cache size influences the miss-rate and therefore the amount of memory accesses, this affects the impact of the modules as well. Aside from the 4kb caches from the previous section, the same measurements were performed with caches of 2kb, 8kb and 16kb in size.

6.3.1. Baseline

The baseline performance of the benchmarks and caches without EMS modules was determined first. Figure 6.8a shows the miss-rates of the data cache for all benchmarks and cache sizes. The miss-rates were determined by counting the total amount of cache requests from the processor and how many of these resulted in a $MISS$ state. Here, the left bars show the cold-start measurements and the right bars represent the average over 9 subsequent runs. The instruction cache for all benchmarks only showed a miss-rate of less than 0.5% during the cold-start run, and subsequent runs saw zero misses no matter the cache size. Figure 6.8b shows the actual execution times of the benchmark runs. Here, the left bars represent the cold-start, and the right bars the average over 10 runs. As before, no averages were determined for the Qsort benchmark. For clarity these same times are shown in Table 6.2. Some trends visible in the benchmark performance will be covered next.



(a) Data cache miss-rates. Left bars represent cold-start, right bars show subsequent runs.

(b) Benchmark execution times. Left bars represent cold-start, right bars show average over 10 runs.

Figure 6.8: Benchmark performance on baseline platform for all five benchmarks, on all four cache sizes.

Table 6.2: Baseline benchmark execution times in μs for each cache size.

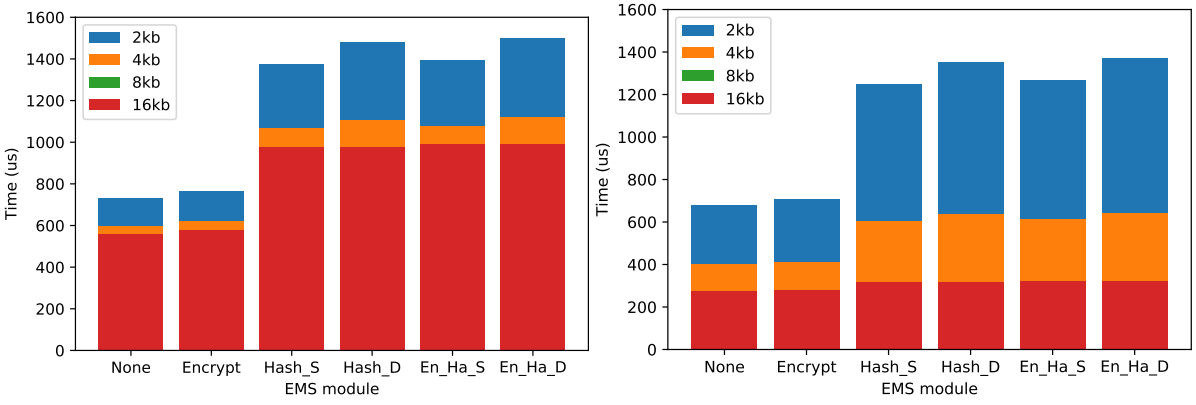
Benchmark	Run	2kb	4kb	8kb	16kb
Median	Cold-start	732	597	557	557
	Average	678	401	277	277
Multiply	Cold-start	864	864	864	864
	Average	751	751	751	751
Qsort	Cold-start	10965	9606	7588	6625
	Average	N/A	N/A	N/A	N/A
Towers	Cold-start	2880	2880	2880	2880
	Average	2811	2811	2811	2811
Vvadd	Cold-start	605	481	467	467
	Average	564	308	172	172

The Median and Vvadd benchmarks show similar behaviour: The smallest caches of 2kb lead to a high cold-start miss-rate of 13 and 20 percent respectively, which only slightly decreases during subsequent runs. A larger cache of 4kb, on the other hand, reduces the miss-rate significantly. This effect is amplified for runs following the cold-start. Increasing the cache-size further leads to diminishing returns during the first run, although it does completely prevent any cache-misses during the subsequent runs. These results are also visible in the execution times. Larger caches result in higher performance over multiple runs, though increasing them beyond 8kb does not offer any additional advantage. Cold-start performance is increased by 31 and 25 percent at most, over the smallest 2kb cache for Median and Vvadd respectively.

Multiply and Towers, too, show similar behaviour to each other. Though Multiply shows a 17% miss-rate during the cold-start run, this percentage is not affected at all by cache-sizes ranging from 2kb to 16kb. After this initial run, its entire dataset is stored in cache, and subsequent runs do not lead to any new misses. As Towers does not have a dataset at all, it does not suffer from this cold-start penalty and shows no data cache-misses for any runs. Increasing the cache size also does not lead to significant improvements in execution time, with the smallest caches performing just as well as the largest ones.

Finally, the Qsort benchmark with its large dataset and complicated algorithm, shows a slightly different result. Increasing the cache-size shows a more gradual improvement in performance. The miss-rate drops from 7.5% with the 2kb caches to 6.2%, 3.1% and 2.5% for the $4kb$, 8kb and 16kb caches respectively. The same is visible in the execution times, which gradually drop for larger caches. Due to the reasons stated above, no data regarding average runs was recorded.

6.3.2. Median



(a) Cache cold-start

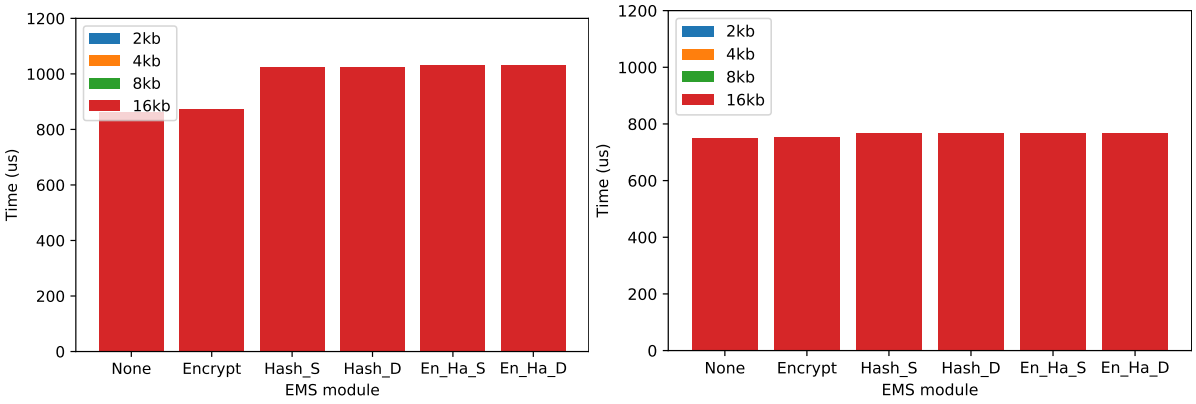
(b) 10 runs average

Figure 6.9: Execution times of Median benchmark for 4 cache sizes.

Figure 6.9 presents the execution times of the Median benchmark, for all four cache sizes. It shows that the smallest 2kb cache with more cache-misses leads a slower execution. This effect is amplified by the integrity-verifying EMS modules, nearly doubling the execution times. Increasing the cache-size

shows clear speedups, in particular during runs after the cold-start, though no more improvements are gained beyond 8kb. At that size, the integrity-verifying modules result in a 75 and 15 percent increase for the cold-start and subsequent runs respectively.

6.3.3. Multiply



(a) Cache cold-start

(b) 10 runs average

Figure 6.10: Execution times of Multiply benchmark for 4 cache sizes.

The effects, or lack thereof, of the cache-size on the execution of the Multiply benchmark can be seen in Figure 6.10. No changes are visible when larger caches are used. During cold-starts, the hashing variants of the EMS module cause a 19% increase in execution time, which falls to 2% over 10 subsequent runs.

6.3.4. Qsort

Figure 6.11 shows the results for the Qsort benchmark. Larger caches have a clear effect on the performance of this benchmark, in particular when integrity verification is performed. For the 2kb caches, the platform with this functionality is close to twice as slow as the baseline. At 16kb, this impact is limited to 20% at worst.

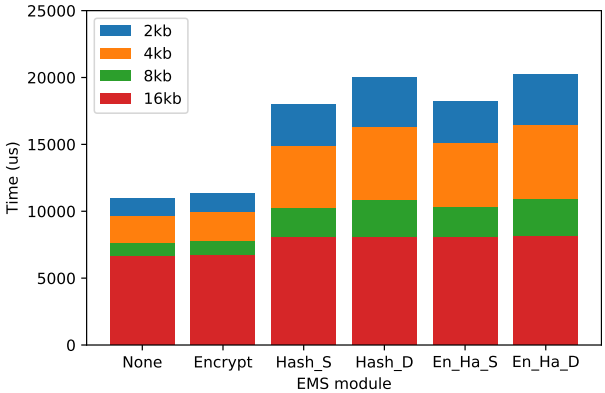
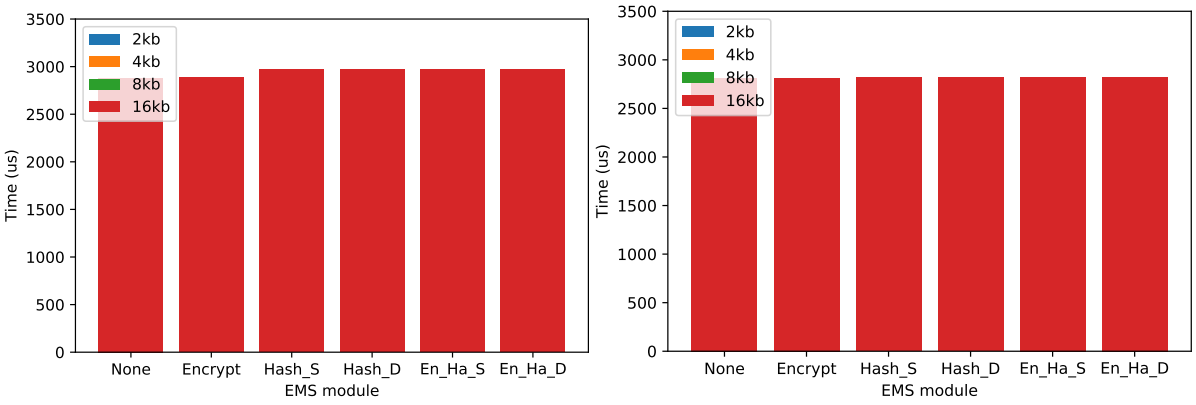


Figure 6.11: Execution times of Qsort benchmark for 4 cache sizes.

6.3.5. Towers

As above in Section 6.2, the EMS modules have no significant influence on the execution times of the Towers benchmark. For completeness, Figure 6.12 shows the performance for all four cache-sizes. Without a dataset however, no differences are visible.

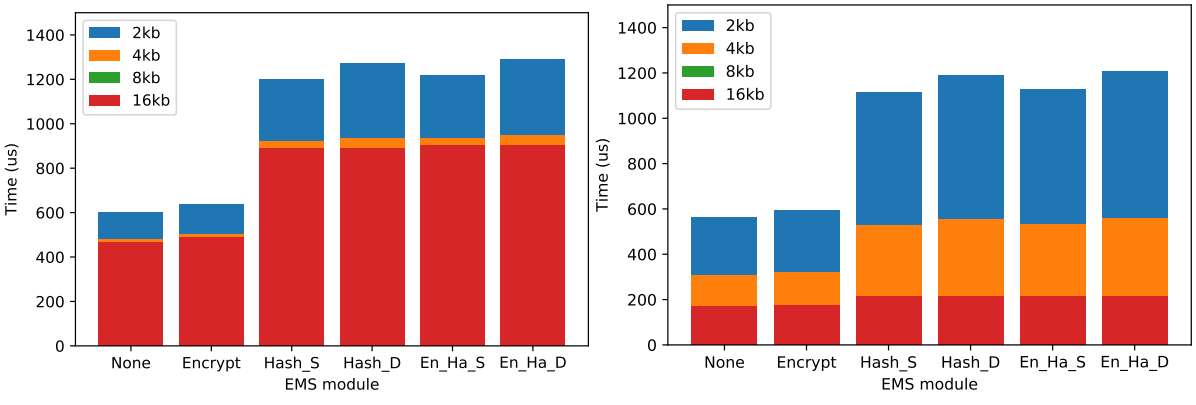


(a) Cache cold-start (b) 10 runs average

Figure 6.12: Execution times of Towers benchmark for 4 cache sizes.

6.3.6. Vvadd

Figure 6.13, finally, displays the results for the Vvadd benchmark. As in Section 6.2, the behaviour is similar to that of the Median application. Increasing the cache-size beyond 2kb shows large performance improvements, which stop at 8kb. At that point, the hashing EMS modules cause a 90% degradation in performance during the initial cold-start, dropping to 25% on average over 10 runs.



(a) Cache cold-start (b) 10 runs average

Figure 6.13: Execution times of Vvadd benchmark for 4 cache sizes.

6.3.7. Total

To support the figures from the previous sections, Table 6.3 shows the same results numerically. It provides the execution times for all benchmarks, with all EMS module variants and all four cache sizes. The percentages represent the increase in execution times, relative to the baseline figures with the same cache size.

Table 6.3: Benchmark execution times for each module variant and cache size.

		Baseline				Encrypt			
		2kb	4kb	8kb	16kb	2kb	4k	8kb	16kb
Median	C	732	597	557	557	767 (4.7%)	622 (4.2%)	580 (4.2%)	580 (4.2%)
	A	678	401	277	277	709 (4.6%)	412 (2.7%)	279 (0.7%)	279 (0.7%)
Multiply	C	864	864	864	864	873 (1.0%)	873 (1.0%)	873 (1.0%)	873 (1.0%)
	A	751	751	751	751	752 (0.1%)	752 (0.1%)	752 (0.1%)	752 (0.1%)
Qsort	C	10965	9606	7588	6625	11345 (3.5%)	9891 (2.9%)	7731 (1.9%)	6701 (1.1%)
	A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Towers	C	2880	2880	2880	2880	2885 (0.2%)	2885 (0.2%)	2885 (0.2%)	2885 (0.2%)
	A	2811	2811	2811	2811	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)	2812 (0.1%)
Vvadd	C	605	481	467	467	635 (4.9%)	504 (4.7%)	489 (4.7%)	489 (4.7%)
	A	564	308	172	172	593 (5.1%)	320 (3.9%)	174 (1.2%)	174 (1.2%)
		MAC S				MAC D			
		2kb	4kb	8kb	16kb	2kb	4k	8kb	16kb
Median	C	1374 (87.7%)	1066 (78.5%)	976 (75.2%)	976 (75.2%)	1479 (102.0%)	1105 (85.1%)	976 (75.2%)	976 (75.2%)
	A	1250 (84.3%)	606 (51.1%)	319 (15.2%)	319 (15.2%)	1355 (99.8%)	638 (59.1%)	319 (15.6%)	319 (15.6%)
Multiply	C	1026 (18.7%)	1026 (18.7%)	1026 (18.7%)	1026 (18.7%)	1026 (18.8%)	1026 (18.8%)	1026 (18.8%)	1026 (18.8%)
	A	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)	767 (2.1%)
Qsort	C	17998 (64.1%)	14874 (54.8%)	10229 (34.8%)	8028 (21.2%)	20000 (82.4%)	16265 (69.3%)	10786 (42.1%)	8038 (21.3%)
	A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Towers	C	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)	2971 (3.2%)
	A	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)	2820 (0.3%)
Vvadd	C	1199 (98.1%)	920 (91.2%)	888 (90.1%)	888 (90.1%)	1270 (109.9%)	933 (93.9%)	888 (90.1%)	888 (90.1%)
	A	1112 (97.1%)	527 (71.1%)	214 (24.4%)	214 (24.4%)	1189 (110.8%)	554 (79.9%)	214 (24.4%)	214 (24.4%)
		Enc + MAC S				Enc + MAC D			
		2kb	4kb	8kb	16kb	2kb	4k	8kb	16kb
Median	C	1394 (90.4%)	1080 (80.9%)	989 (77.5%)	989 (77.5%)	1499 (104.7%)	1119 (87.4%)	989 (77.5%)	989 (77.5%)
	A	1268 (87.0%)	612 (52.6%)	320 (15.0%)	320 (15.0%)	1372 (102.3%)	644 (60.6%)	320 (15.5%)	320 (15.5%)
Multiply	C	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)	1031 (19.3%)
	A	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.3%)	768 (2.2%)	768 (2.2%)	768 (2.2%)	768 (2.2%)
Qsort	C	18215 (66.1%)	15036 (56.3%)	10310 (35.9%)	8071 (21.8%)	20217 (84.3%)	16428 (71.0%)	10867 (43.2%)	8081 (21.9%)
	A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Towers	C	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)	2973 (3.2%)
	A	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)	2821 (0.3%)
Vvadd	C	1217 (101.2%)	934 (94.2%)	901 (92.9%)	901 (92.9%)	1288 (112.8%)	947 (96.9%)	901 (92.9%)	901 (92.9%)
	A	1129 (100.1%)	533 (73.1%)	215 (25.0%)	215 (25.0%)	1206 (113.8%)	560 (81.8%)	215 (25.0%)	215 (25.0%)

6.4. Security Evaluation

Next, the security provided by the EMS modules was tested by performing attacks on the platform. This section covers three types as explained in Section 6.1: Injections, rogue memory and replay attacks. For each attack the results of the baseline platform, the encryption EMS variant and the hashing EMS module are compared.

6.4.1. Fault/Code Injection

Fault injections aim to flip one or more bits in memory to induce unintended behaviour. Code injections are similar, where instructions or even full applications are changed into other ones. As such, both attacks can be simulated by changing one or more bits in a location in memory, that is read during execution of an application. In the following examples, the last bit at address 42 in the instruction memory was randomly chosen and changed, with the platform running the Median benchmark. The same was done for 9 other random locations in memory, leading to identical results.

Baseline

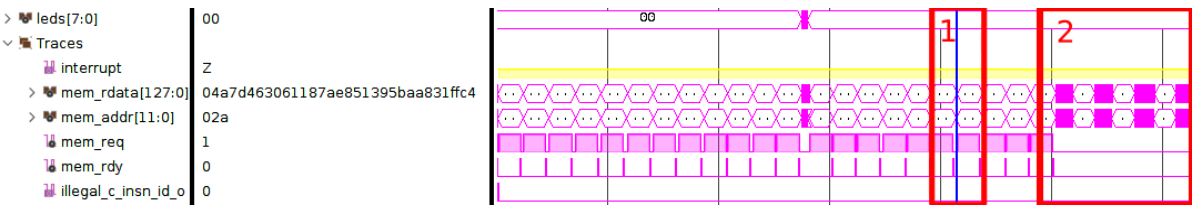


Figure 6.14: Fault injection attack performed on baseline platform

Figure 6.14 shows the result of the injection on the baseline platform with no EMS module. The altered memory line is being read in the area marked with 1. In this case no illegal instruction follows and the processor continues with unintended behaviour, marked by area 2 and onwards. In fact, it starts

reading and writing seemingly random data to and from its data memory, never returning to normal operation.

Encrypt

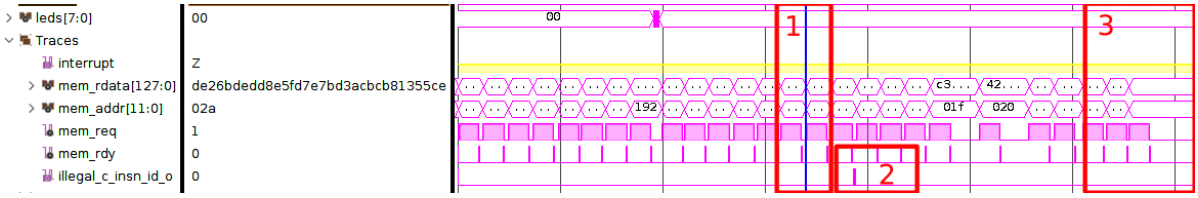


Figure 6.15: Fault injection attack performed on platform with encrypting EMS module

Figure 6.15 displays the result when memory is encrypted by an EMS module. The affected line is being read in the area marked with 1. Decryption leads to entirely different data, causing an illegal instruction error in area 2. From area 3 and onwards, the processor has crashed and no more activity is observed.

MAC

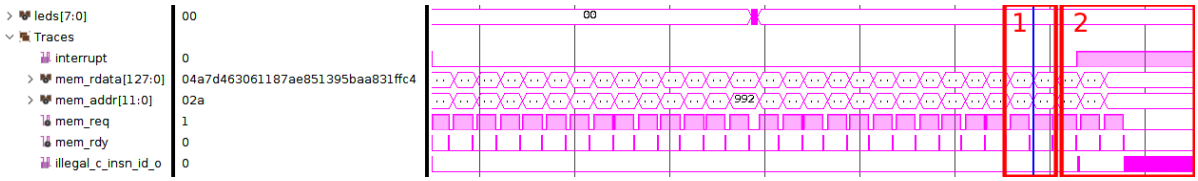


Figure 6.16: Fault injection attack performed on platform with hashing EMS module

Figure 6.16 shows what happens when memory integrity is verified by and EMS module. Area 1 shows the altered line being read, immediately followed by an interrupt in area 2 as the MACs do not match. From this point onwards, the EMS module only presents zeros to the processor and activity halts.

6.4.2. Rogue Memory

Rogue memory attacks are when an attacker changes the entire contents of a memory chip with those of another, or physically removes the original chip and installs the memory chip of another unit to a device. This way, one device can be 'borrowed' from a target and extensively researched for vulnerabilities. If it turns out that any issues that are found have already been patched by updates for the other devices, the attacker could perform a rogue-memory attack to 'restore' its target to a previous firmware version. In this section, a rogue-memory situation is emulated by loading a benchmark that was secured by the key of another device into the platform, and simulating the results.

Baseline

As there is no security available in the baseline platform, performing a rogue-memory attack is trivial. The processor simply executed the benchmark without any issues. An attacker would be able to change the contents of the memory to anything he wants, as long as it is a valid application that can be executed.

Encrypt

In case the contents of memory are encrypted, a rogue-memory attack will not work as intended. Even if the attacker is advanced and funded enough to extract the secret key of his 'borrowed' research device, the target will be using a different key. When decrypting the new memory contents with another key, the result will be the same as for the injection attack in Figure 6.15. Invalid instructions are encountered and the processor crashes.

MAC
Similar to above, a device's memory contents are secured with a unique cryptographic key. As such, replacing the memory with a chip or contents from another device, will lead to mismatching MAC values. These are detected at the very first read from memory, and the processor will hang as in the injection case from Figure 6.16.

6.4.3. Replay Attacks

Replay attacks are performed by replacing data in memory with values that were stored earlier or in a different location in the same memory. This type of attack remains a threat to the developed EMS modules as no time or address-based verification is implemented. For completeness, a replay attack was performed on the platform with and without EMS modules. In the following examples, address 42 in the instruction memory was again modified: Its contents were replaced with those of address 43.

Baseline

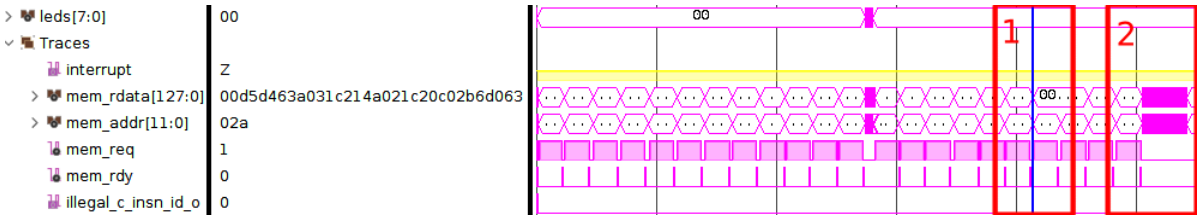


Figure 6.17: Replay attack performed on baseline platform

Figure 6.17 shows the results of the baseline platform when a replay attack is performed. Area 1 shows the affected memory line being read, with the platform getting stuck in a loop in area 2 and onwards. The result is similar to that of code injection, as all 'new' instructions are valid and the processor does not detect anything being wrong.

Encrypt

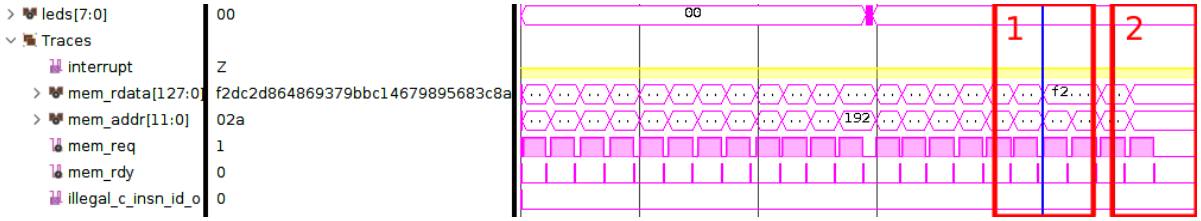


Figure 6.18: Replay attack performed on platform with encrypting EMS module

Figure 6.18 displays the result when and EMS module has encrypted all memory contents. In this case the ciphertext data in line 42 is replaced with the ciphertext data from line 43. The affected line is being read in area 1, after which it is decrypted into valid instructions identical to those in the baseline case. From area 2 onwards, the processor shows identical behaviour to the baseline version. Though the mem_rdata and mem_addr traces appear different from the baseline, this is because now the EMS module is located between the cache and RAM, blocking any changes to the cache's address output with no valid memory request.

MAC
Figure 6.19 shows the results when only the data part is replaced in a replay attack, with an integrity-verifying EMS module present. The data again is being read in area 1. As now the MAC does not match, the module will produce an interrupt and behave similar to an injection attack as shown in area 2. If an attacker is able to replace both the data and its corresponding MAC in memory, the result is shown in Figure 6.20. Now, the module does not detect a mismatched MAC and the result is the same as for the baseline platform.

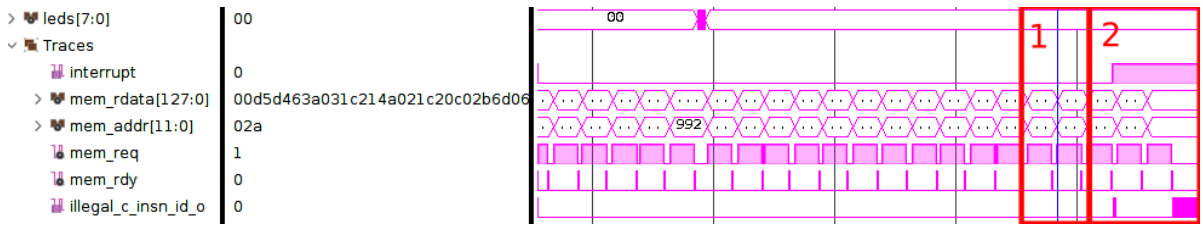


Figure 6.19: Replay attack performed on platform with hashing EMS module, not including hash

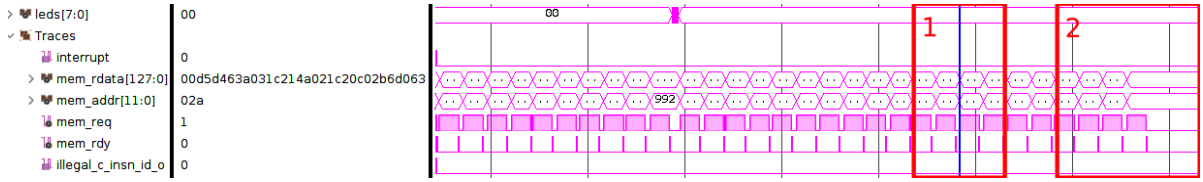
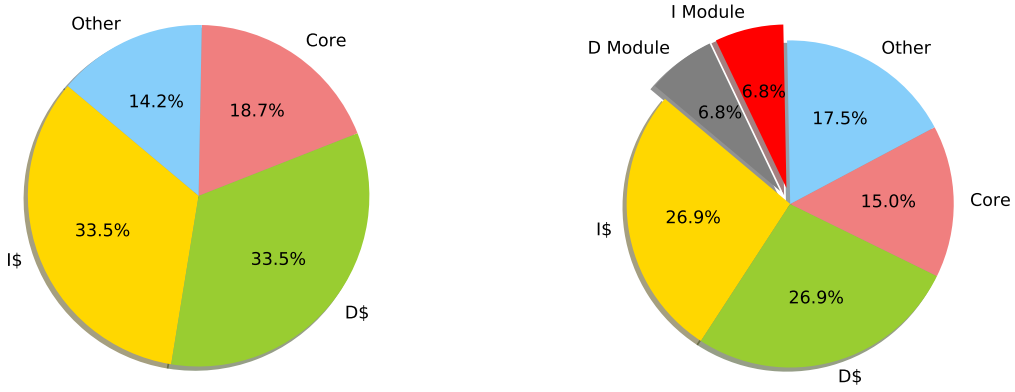


Figure 6.20: Replay attack performed on platform with hashing EMS module, including hash

6.5. Hardware Overhead Evaluation

To measure the hardware overhead of the EMS modules, they were added to the development platform and synthesized in Vivado. As mentioned in Section 6.1; due to size constraints of the available PYNQ FPGA board, the platform could only be synthesized with 2kb, 16-way data and instruction caches. The target clock frequency was set at 25MHz and the baseline platform without module was synthesized first. Its resulting timing and area constraints can be seen in Table 6.4, as well as schematically in Figure 6.21a for clarity. In this configuration, the platform requires more than 60% of the FPGA's available LUTs, two-thirds of which is taken by the combined data and instruction caches. The actual processor core itself is responsible for 19% of the area taken by the platform. At 25MHz, the estimated Worst Negative Slack (WNS) was found to be nearly 5ns, meaning the frequency could potentially be raised to a maximum of 28.5MHz under the same conditions.



(a) Baseline (b) With largest module

Figure 6.21: FPGA area distribution of baseline platform and platform with largest module

Table 6.5 shows the additional area requirements of each variant of the EMS module, as well as their impact on the Worst Negative Slack. The encryption functionality is more expensive than hashing when it comes to area. This is true for both the single and double hash configurations. An encrypting EMS module on its own takes up one third as much area as the RI5CY core, where the hashing variants require just over a quarter. Combining these functionalities in one EMS module costs less than its sum, with less than half the area of the core. The module with encryption and double hashes represents the worst-case, taking up 45% the amount of LUTs as the RI5CY core. This effect is visualized in Figure 6.21b. Finally, the module variants with a single hash per memory line require half of the available memory to store their tags, whereas the optimized double hash variants take up one third.

All versions of the EMS module could synthesize successfully along with the platform at the set

Table 6.4: Hardware requirements of baseline platform with 2kb, 16-way caches.

	Slice LUTs (Of 53200)	Slice Regs (Of 106400)	WNS @ 25MHz
Full platform	32057	43871	4.97ns
RI5CY core	6006	2203	
Cache (x2)	10752	19019	

Table 6.5: Hardware requirements of platform with modules and 2kb, 16-way caches.

	Slice LUTs	Slice Regs	WNS @ 25MHz	RAM capacity lost to MAC
Encrypt				
Full platform	35176, +9.7%	45025, +2.6%	4.30ns	0.0%
EMS module (x2)	2090	560		
Prince core	1905	130		
MAC Single				
Full platform	34003, +6.1%	46056, +4.9%	3.88ns	50.0%
EMS module (x2)	1505	1084		
SipHash core	863	467		
MAC Double				
Full platform	34197, +6.7%	46461, +5.9%	4.38ns	33.3%
EMS module (x2)	1596	1277		
SipHash core	851	467		
Encrypt+MAC (S)				
Full platform	36126, +12.6%	46589, +6.2%	4.21ns	50.0%
EMS module (x2)	2563	1346		
Prince core	1526	130		
SipHash core	762	467		
Encrypt+MAC (D)				
Full platform	36427, +13.6%	46983, +7.1%	1.75ns	33.3%
EMS module (x2)	2713	1539		
Prince core	1590	130		
SipHash core	847	467		

25MHz clock frequency. The WNS was reduced somewhat compared to the base implementation without module. Here, too, the module with encryption and double hashes saw the largest impact with an estimated maximum frequency of 26.3MHz, a drop of 2.2MHz over de baseline. It should be noted though that no nets related to the modules were ever listed among the slowest paths by Vivado.

6.6. Related Works Comparison

Finally, the cost of the modules is compared to that of that of the art proposals. The first work implemented AES in GCM mode to achieve authenticated encryption [17]. A verilog design of this mode of operation was taken from [112] and synthesized in Vivado as a standalone unit for the same target FPGA. It was not connected to other functionality to get the area requirements. The second work applied the SHA-256 hash function to memory contents related to specific applications [15]. In the same manner, only the core IP is considered for evaluation, in this case the SHA-256 taken from [113]. Results can be seen in Table 6.6, with area requirements compared to the largest of the EMS modules.

From the table it follows that the hardware requirements of the GCM-AES implementation are close to those of the largest EMS module. This is however not the full story, as the EMS module includes all functionality to work in this application. The GCM-AES implementation will still need the additional logic to be used as a memory security module. Furthermore, additional logic is required for the timestamp generation hardware which is needed to make this mode of operation secure. Even more significant, the design presented in [17] requires storing this 32-bit timestamp on-chip, for each address in memory. Combined, the area requirements would be significantly larger than even the largest EMS variant. From the timing side of things; the GCM-AES implementation requires 20 clock cycles to initialize for each

Table 6.6: Hardware requirements comparison to state of the art

	Slice LUTs	Slice Regs	Bram
Encrypt+MAC (D)	2713 (100%)	1539 (100%)	0
Prince core	1590	130	0
SipHash core	847	467	0
GCM-AES [112]	2670 (98%)	1568 (102%)	5
SHA-256 [113]	2027 (75%)	1830 (119%)	0

memory access, followed by 18 cycles to process a single 128-bit block of data. This results in an overhead of 38 cycles, excluding the additional memory access for the tag. Performance here is close to that of the slowest EMS module, coming in at 37 cycles excluding the memory accesses. In a full implementation however, the GCM-AES version would still require additional cycles for the EMS logic and the timestamp generation. Similarly, the SHA-256 implementation is significantly larger than the used SipHash. Including SHA-256 in an EMS modules, would therefore greatly increase the size. Furthermore, the SHA implementation requires 66 cycles to process a single block, twice that of the EMS module with both encryption and MAC calculation.

6.7. Discussion

Based on the experiments, the following can be highlighted:

Security: The modules performed as expected during the attacks in Section 6.4. The MAC variants counter all sorts of hardware based attacks on memory, including fault and code injection, by alerting the processor and halting operation. Although encryption alone does not prevent such attacks from being attempted, it does prevent the attacker from learning what he is actually doing. This alone would make many such attacks impractical. Both options completely counter a rogue-memory attack, as long as each device has access to a unique key. Replay attacks remain a weakness for the current implementations of the EMS modules, and more research will be needed to counter these.

Performance: The performance impact of the modules varies widely per benchmark and per cache-size. Cache miss-rates as high as 20% were observed, whereas in modern systems miss-rates are typically lower than 5% [71]. For runs where these rates were at or below 5%, the slowest version of the module -encrypt + MAC (Double)- caused the execution times to increase by 25% at worst and 0.3% at best. Where the single MAC variant performs noticeably better at the smaller cache sizes, at this point the difference is no longer significant. Compared to the MAC implementations, encryption causes a minimal overhead of less than one percent typically, and less than 5% in the worst case.

Area Overhead: The area overhead of the EMS modules when implemented in hardware caused a less than 14% increase in LUT requirements over the full processor platform. However, this platform required two of the modules, one for each cache which were connected to their own RAM, whereas typically only one would be required. Compared to the used RI5CY microcontroller core, the largest module -encrypt + MAC (Double)- was 45% its size. Compared to a more powerful and perhaps more realistic processor core such as Ariane [82] with its 85000 LUTs requirements, this largest module would cause only 3.2% overhead.

Evaluations under Operating Systems: An attempt was made to implement the EMS modules for the Ariane platform. This would have allowed to measure its effect on a more powerful IoT platform with a real operation system such as Linux, multi-layered caches, flash storage support and enough memory to run larger and custom benchmarks. However, it did take half a year before the order of the required FPGA development board was processed and as such the board arrived too late for this thesis. Porting the EMS modules to this platform is left for future research.

Security Extensions Evaluations: As presented in Chapter 4, the security provided by EMS modules can be increased by including two other techniques, namely a secure network protocol and software binding. Since it was not possible to perform such experiments without having an Operating System, these security extensions could not be implemented and tested. Therefore, we consider this part as future works.

7 Conclusion

This chapter concludes this thesis by providing the conclusions and some points left open for future research. The main conclusions of each chapter are first summarized in Section 7.1, followed by the future research options in Section 7.2.

7.1. Summary

Chapter 1 pointed out the threat of cybercrime, in particular regarding the Internet of Things, which is expected to continue to grow rapidly to make up half of the total internet connections by 2023, according to Cisco. It discussed how IoT devices are already installed in a large variety of applications, some of which pose a considerable risk to their surroundings if the device were to be attacked. Several real-world examples of this were given. The chapter then highlighted hardware-based attacks aimed at the external memory modules of a device, which are insufficiently protected and are an attractive target in high-value applications. The rest of the thesis is focussed on developing memory security modules to counter this threat.

Chapter 2 of this thesis discussed the types of hardware and software that are typically found in IoT devices. Next, it explained the importance of the security criteria integrity, authenticity, confidentiality and availability. It then defined three categories of IoT applications based on the severity of the consequences in case a device is compromised. These categories were low, medium and high risk, with the consequences ranging from personal inconvenience, danger to human life and danger to society on a larger scale. This chapter re-enforces the point that the security of IoT devices is critical.

Chapter 3 discussed three different categories of attacks based on their main attack vector being network based, software based or hardware based. For each category, the workings, some examples and existing countermeasures were covered for multiple different types of attack.

Chapter 4 presented the security solution developed to counter hardware based threats on memory modules. It first laid out the constrained environments of IoT devices to focus on hardware, and then explained the concept and operation of the Embedded Memory Security modules as a solution. Some additional considerations were mentioned and discussed to make the modules more secure. It ended by proposing two security extensions that could be implemented along with the presented EMS modules to also secure against software and network threats.

Chapter 5 covered the design choices made when developing the EMS modules. The chosen development platform, cryptographic cipher and MAC function used in the modules were discussed in detail by covering the requirements, available options and trade-offs as well as the final specifications for each. Then, it mentions the five variants of the EMS module that were developed, their components and their interfaces.

Chapter 6 presented the results of the developed modules. They were integrated into the processor development platform and their performance was determined through various experiments. The setup and performed experiments were first described in detail. Then, the performance impact of the modules was determined by running several benchmarks and testing with multiple cache sizes. This showed that under realistic cache conditions, their impact was limited to a 25% increase in execution time at worst. Following this, three attacks were performed on the platform to observe the security properties of

the modules, demonstrating that they do protect against various hardware attacks on memory. Finally, the hardware cost of the modules was determined and compared to state of the art works. It was found that the most expensive module cost less than half the area of the RI5CY microcontroller core, and only 3% of the linux-capable Ariane core.

7.2. Future work

Finally, this section lists several paths to explore further in future research as follows:

1. **Flash support:** The performance impact of the EMS modules was measured when applied to RAM with multiple benchmarks and multiple cache sizes. Aside from this, the modules could also be used to secure other interfaces such as persistent flash storage, which was originally planned to be part of this thesis. Due to time constraints and the lack of flash support by the platform, this was left for future research.
2. **Operating systems:** The microcontroller-class processor platform did not have an operating system. Implementing the modules in a larger platform such as Ariane, would allow for measuring the effects when an operating system is present. Furthermore, it would provide functionality to run more advanced benchmarks. This, too, was intended to be part of this thesis, however due to development boards not arriving on time this was not possible.
3. **Replay attacks:** Although the developed modules protect against several hardware-attacks on memory, they are unable to detect a replay attack. Some state-of-the-art proposals mentioned in Chapter 1 do offer this functionality, at a significant cost in hardware. If this type of attack is considered relevant for an application, a lightweight solution to this would also be required.
4. **Security extensions:** The security extensions mentioned in Chapter 4 would offer protection against software and network based attacks. Developing these further is left open to future work.
5. **Optimizations:** Additional optimizations could be implemented to the developed modules to save on time and area costs. For example, operations could be parallelized to save clock cycles. Similarly, cache lines that are larger than what is stored at one memory address could be used. Then, a single MAC could be calculated per cache line instead of for each address, reducing the memory requirements. These and other optimizations could be explored.
6. **Tweakable authenticated encryption:** Tweakable ciphers are a recent development, which would allow for using the address as an additional input when encrypting data. This is briefly mentioned in Chapter 4 as a solution against memory patterns, but could also help against replay attacks. Similarly, lightweight implementations of authenticated encryption could be further developed to combine encryption and MAC calculation into one operation.

Bibliography

[1] "Cisco Annual Internet Report (2018–2023) White Paper," Cisco, Tech. Rep., 03 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[2] "The Economic Impact of Cybercrime - No Slowing Down," McAfee, Tech. Rep., 02 2018. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/executive-summaries/es-economic-impact-cybercrime.pdf

[3] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 1093–1110.

[5] C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle," Tech. Rep., 08 2015. [Online]. Available: http://illmatics.com/Remote%20Car%20Hacking.pdf

[6] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, K. Fu, and D. Song, "Take two software updates and see me in the morning: the case for software security evaluations of medical devices," 08 2011, pp. 6–6.

[7] A. Barenghi, G. M. Bertoni, L. Breveglieri, M. Pellicioli, and G. Pelosi, "Low voltage fault attacks to aes," in *2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2010, pp. 7–12.

[8] N. Timmers, A. Spruyt, and M. Witteman, "Controlling pc on arm using fault injection," in *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Aug 2016, pp. 25–35.

[9] G. Hernandez and D. Buentello, "Smart nest thermostat a smart spy in your home," 2014.

[10] M. Hoekstra. Intel® sgx for dummies (intel® sgx design objectives). [Online]. Available: https://software.intel.com/content/www/us/en/develop/blogs/protecting-application-secrets-with-intel-sgx.html

[11] P. A. H. Peterson, "Cryptkeeper: Improving security with encrypted ram," in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, Nov 2010, pp. 120–126.

[12] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "Trustzone explained: Architectural features and use cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Nov 2016, pp. 445–451.

[13] S. Agrawal, M. Das, A. Mathuria, and S. Srivastava, "Program integrity verification for detecting node capture attack in wireless sensor network," 12 2015, pp. 419–440.

[14] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Workshop on Wireless Security*, 2006.

[15] G. E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 339–350.

[16] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez, "A parallelized way to provide data encryption and integrity checking on a processor-memory bus," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 506–509.

[17] J. Crenne, R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier, and D. Unnikrishnan, "Configurable memory security in embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, 03 2013.

[18] M. O. Ojo, S. Giordano, G. Procissi, and I. N. Seitanidis, “A review of low-end, middle-end, and high-end iot devices,” *IEEE Access*, vol. 6, pp. 70 528–70 554, 2018.

[19] Moteiv Corporation, “Tmote sky datasheet,” 2005. [Online]. Available: https://fccid.io/TOQTMOTESKY/User-Manual/Users-Manual-Revised-613136

[20] S. Checkoway, D. Mccoy, D. Anderson, B. Kantor, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, “Comprehensive experimental analyses of automotive attack surfaces,” 08 2011.

[21] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, “Riot: An open source operating system for low-end embedded devices in the iot,” *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[22] ISO 27000:2018(E), “Information technology — Security techniques — Information security management systems — Overview and vocabulary,” International Organization for Standardization, Geneva, CH, Standard, Feb. 2018.

[23] O. Adeyinka, “Internet attack methods and internet security technology,” in *2008 Second Asia International Conference on Modelling Simulation (AMS)*, May 2008, pp. 77–82.

[24] I. Andrea, C. Chrysostomou, and G. Hadjichristofi, “Internet of things: Security vulnerabilities and challenges,” in *2015 IEEE Symposium on Computers and Communication (ISCC)*, July 2015, pp. 180–187.

[25] N. P. Smart, *Cryptography Made Simple*. Springer, Cham, 2016. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-21936-3#toc

[26] A. Cui, M. Costello, and S. Stolfo, “When firmware modifications attack: A case study of embedded exploitation,” 02 2013.

[27] Rapid7 corporation. (2015) Hacking iot: A case study on baby monitor exposures and vulnerabilities. [Online]. Available: https://www.rapid7.com/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf

[28] D. Berkowitz. (2011) Lg smart refrigerator. [Online]. Available: https://en.wikipedia.org/wiki/Smart_refrigerator#/media/File:LG_Smart_Refrigerator_at_CES_2011.jpg

[29] P. Beltrán-García, E. Aguirre-Anaya, P. J. Escamilla-Ambrosio, and R. Acosta-Bermejo, “Iot botnets,” in *Telematics and Computing*, M. F. Mata-Rivera, R. Zagal-Flores, and C. Barría-Huidobro, Eds. Cham: Springer International Publishing, 2019, pp. 247–257.

[30] S. Sciancalepore, G. Oligeri, and R. Pietro, “Strength of crowd (soc)-defeating a reactive jammer in iot with decoy messages,” *Sensors*, vol. 18, 10 2018.

[31] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, “Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses,” in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008, pp. 129–142.

[32] K. Zetter, “Inside the cunning, unprecedented hack of ukraine’s power grid,” *Wired*. [Online]. Available: https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/

[33] “Obama says u.s. has asked iran to return drone aircraft,” *CNN*. [Online]. Available: https://edition.cnn.com/2011/12/12/world/meast/iran-us-drone/index.html

[34] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Sunnyvale, CA, USA: Insecure, 2009.

[35] O. Solon, “Cyber-extortionists targeting the financial sector are demanding bitcoin ransoms,” *Bloomberg*. [Online]. Available: https://www.bloomberg.com/news/articles/2015-09-09/bitcoin-ddos-ransom-demands-raise-dd4bc-profile?mod=djemRiskCompliance

[36] P. Association, “Anonymous claims responsibility for taking down government sites,” *The Guardian*. [Online]. Available: https://www.theguardian.com/technology/2012/apr/08/anonymous-taking-down-government-websites

[37] M. Anirudh, S. A. Thileeban, and D. J. Nallathambi, “Use of honeypots for mitigating dos attacks targeted on iot networks,” in *2017 International Conference on Computer, Communication and Signal Processing (ICCCSP)*, Jan 2017, pp. 1–4.

[38] N. Namvar, W. Saad, N. Bahadori, and B. Kelley, “Jamming in the internet of things: A game-theoretic perspective,” in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[39] A. Gallais, T. Hedli, V. Loscri, and N. Mitton, “Denial-of-sleep attacks against iot networks,” in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2019, pp. 1025–1030.

[40] K. Zhang, X. Liang, R. Lu, and X. Shen, “Sybil attacks and their defenses in the internet of things,” *IEEE Internet of Things Journal*, vol. 1, no. 5, pp. 372–383, 2014.

[41] D. Papp, Z. Ma, and L. Buttyan, “Embedded systems security: Threats, vulnerabilities, and attack taxonomy,” in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, July 2015, pp. 145–152.

[42] S. Hansman and R. Hunt, “A taxonomy of network and computer attacks,” *Computers & Security*, vol. 24, no. 1, pp. 31 – 43, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404804001804

[43] B. Zhu, A. Joseph, and S. Sastry, “A taxonomy of cyber attacks on scada systems,” in *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, Oct 2011, pp. 380–388.

[44] “Common vulnerabilities and exposures database.” MITRE. [Online]. Available: http://cve.mitre.org/

[45] K. Tsipenyuk, B. Chess, and G. McGraw, “Seven pernicious kingdoms: a taxonomy of software security errors,” *IEEE Security Privacy*, vol. 3, no. 6, pp. 81–84, Nov 2005.

[46] A. Mohanty, I. Obaidat, F. Yilmaz, and M. Sridhar, “Control-hijacking vulnerabilities in iot firmware: A brief survey,” 04 2018.

[47] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.” in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[48] P. Mackenzie. (2019) Wannacry aftershock. SophosLabs. [Online]. Available: https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/WannaCry-Aftershock.pdf

[49] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, “Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures,” *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov 2012.

[50] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, “The sorcerer’s apprentice guide to fault attacks,” *Proceedings of the IEEE*, vol. 94, no. 2, pp. 370–382, Feb 2006.

[51] M. Hutter, J. Schmidt, and T. Plos, “Contact-based fault injections and power analysis on rfid tags,” in *2009 European Conference on Circuit Theory and Design*, Aug 2009, pp. 409–412.

[52] J. Balasch, B. Gierlichs, and I. Verbauwhede, “An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus,” in *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, Sep. 2011, pp. 105–114.

[53] S. Govindavajhala and A. W. Appel, “Using memory errors to attack a virtual machine,” in *2003 Symposium on Security and Privacy, 2003.*, May 2003, pp. 154–165.

[54] T. J. O’Gorman, “The effect of cosmic rays on the soft error rate of a dram at ground level,” *IEEE Transactions on Electron Devices*, vol. 41, no. 4, pp. 553–557, April 1994.

[55] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, “Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller,” 02 2014.

[56] S. Ordas, L. Guillaume-Sage, and P. Maurine, “Electromagnetic fault injection : the curse of flip-flops,” *Journal of Cryptographic Engineering*, vol. 7, 03 2016.

[57] S. P. Skorobogatov and R. J. Anderson, “Optical fault induction attacks,” in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12.

[58] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 361–372, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2678373.2665726

[59] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397.

[60] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[61] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The em side—channel(s)," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 29–45.

[62] K. Tiri and I. Verbauwhede, "A digital design flow for secure integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1197–1208, July 2006.

[63] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug 2014.

[64] Y. Jin, "Introduction to hardware security," *Electronics*, vol. 4, pp. 763–784, 10 2015.

[65] H. Li, Q. Liu, and J. Zhang, "A survey of hardware trojan threat and defense," *Integration*, vol. 55, pp. 426 – 437, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926016000067

[66] Xiaoxiao Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, June 2008, pp. 15–19.

[67] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[68] C. Stephan Brosnan. (2007) Csiro scienceimage 3876 a remote sensing node part of csiros fleck wireless sensor network technology. [Online]. Available: http://www.scienceimage.csiro.au/image/3876

[69] Maxim integrated, "Maxq1061 datasheet," 12 2019, rev. 5. [Online]. Available: https://www.maximintegrated.com/en/products/microcontrollers/MAXQ1061.html#modalDatasheet

[70] J. Deogirikar and A. Vidhate, "Security attacks in iot: A survey," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, Feb 2017, pp. 32–37.

[71] D. Patterson and J. Hennessy, *Computer Organization and Design*, 5th ed. 225 Wyman Street, Waltham, MA 02451, USA: Morgan Kaufmann, 9 2013, MIPS Edition.

[72] T. McGrath, I. E. Bagci, Z. M. Wang, U. Roedig, and R. J. Young, "A puf taxonomy," *Applied Physics Reviews*, vol. 6, no. 1, p. 011303, 2019. [Online]. Available: https://doi.org/10.1063/1.5079407

[73] U. Rührmair and D. E. Holcomb, "Pufs at a glance," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[74] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," *Journal of Cryptology*, vol. 21, pp. 469–491, 10 2008.

[75] P. Rogaway, M. Bellare, and J. Black, "Ocb: A block-cipher mode of operation for efficient authenticated encryption," vol. 6, no. 3, p. 365–403, Aug. 2003. [Online]. Available: https://doi.org/10.1145/937527.937529

[76] M. Dworkin, "Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality," National Institute of Standards and Technology, Tech. Rep., 2004.

[77] D. Mcgrew and J. Viega, "The galois/counter mode of operation (gcm)," 02 2004.

[78] F. Valsorda. (2013) The ecb penguin. [Online]. Available: https://blog.filippo.io/the-ecb-penguin/

[79] M. Liskov, R. L. Rivest, and D. Wagner, "Tweakable block ciphers," in *Advances in Cryptology — CRYPTO 2002*, M. Yung, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 31–46.

[80] W. Diffie and M. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, November 1976.

[81] *PYNQ-Z1 Board Reference Manual*, Digilent, 4 2017. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual

[82] J. Balkind, “Openpiton + ariane : The first open-source , smp linux-booting risc-v system scaling from one to many cores,” 2019.

[83] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[84] Pulpissimo platform repository. [Online]. Available: https://github.com/pulp-platform/pulpissimo

[85] *Genesys 2 FPGA Board Reference Manual*, Digilent, 8 2017. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/genesys-2/reference-manual

[86] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, “Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct 2017.

[87] *RI5CY: User Manual*, OpenHW Group, 4 2019, rev. 4. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[88] B. Esham. (2007) The computation of a cbc-mac from message blocks m1…mx, with secret key k and block cipher e. [Online]. Available: https://en.wikipedia.org/wiki/CBC-MAC#/media/File:CBC-MAC_structure_(en).svg

[89] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Keccak sponge function family main document,” *Submission to NIST (Round 2)*, vol. 3, no. 30, pp. 320–337, 2009.

[90] J. Guo, T. Peyrin, and A. Poschmann, “The photon family of lightweight hash functions,” in *Advances in Cryptology – CRYPTO 2011*, P. Rogaway, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–239.

[91] A. Biryukov and L. Perrin, “State of the art in lightweight symmetric cryptography,” *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 511, 2017.

[92] S. Badel, N. Dağtekin, J. Nakahara, K. Ouafi, N. Reffé, P. Sepehrdad, P. Sušil, and S. Vaudenay, “Armadillo: A multi-purpose cryptographic primitive dedicated to hardware,” in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 398–412.

[93] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, “spongent: A lightweight hash function,” in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 312–325.

[94] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, “Present: An ultra-lightweight block cipher,” in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 450–466.

[95] T. P. Berger, J. D’Hayer, K. Marquet, M. Minier, and G. Thomas, “The gluon family: A lightweight hash function family based on fcsrs,” in *Progress in Cryptology - AFRICACRYPT 2012*, A. Mitrokotsa and S. Vaudenay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 306–323.

[96] J.-P. Aumasson and D. J. Bernstein, “Siphash: a fast short-input prf,” in *International Conference on Cryptology in India*. Springer, 2012, pp. 489–508.

[97] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, “Chaskey: An efficient mac algorithm for 32-bit microcontrollers,” in *Selected Areas in Cryptography – SAC 2014*, A. Joux and A. Youssef, Eds. Cham: Springer International Publishing, 2014, pp. 306–323.

[98] J. Daor, J. Daemen, and V. Rijmen, “Aes proposal: rijndael,” 10 1999.

[99] M. Hell, T. Johansson, and W. Meier, “Grain: a stream cipher for constrained environments,” *IJWMC*, vol. 2, no. 1, pp. 86–93, 2007.

[100] C. De Cannière, “Trivium: A stream cipher construction inspired by block cipher design principles,” in *Information Security*, S. K. Katsikas, J. López, M. Backes, S. Gritzalis, and B. Preneel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 171–186.

[101] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and Y. Papaefstathiou, “A survey of lightweight stream ciphers for embedded systems,” *Security and Communication Networks*, vol. 9, no. 10, pp. 1226–1246, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1399

[102] G. Hatzivasilis, K. Fysarakis, I. Papaefstathiou, and C. Manifavas, “A review of lightweight block ciphers,” *Journal of Cryptographic Engineering*, vol. 8, no. 2, pp. 141–184, 2018.

[103] C. H. Lim and T. Korkishko, “mcrypton – a lightweight block cipher for security of low-cost rfid tags and sensors,” in *Information Security Applications*, J.-S. Song, T. Kwon, and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 243–258.

[104] K. Shibutani, T. Isobe, H. Hiwatari, A. Mitsuda, T. Akishita, and T. Shirai, “Piccolo: An ultra-lightweight blockcipher,” in *Cryptographic Hardware and Embedded Systems – CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 342–357.

[105] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçın, “Prince – a low-latency block cipher for pervasive computing applications,” in *Advances in Cryptology – ASIACRYPT 2012*, X. Wang and K. Sako, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 208–225.

[106] W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede, “Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms,” *Science China Information Sciences*, vol. 58, 11 2015.

[107] S. Banik, A. Bogdanov, and F. Regazzoni, “Exploring energy efficiency of lightweight block ciphers,” in *Selected Areas in Cryptography – SAC 2015*, O. Dunkelman and L. Keliher, Eds. Cham: Springer International Publishing, 2016, pp. 178–194.

[108] J. Jean, I. Nikolić, T. Peyrin, L. Wang, and S. Wu, “Security analysis of prince,” in *Fast Software Encryption*, S. Moriai, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 92–111.

[109] J. Harttung. Prince vhdl implementation. [Online]. Available: https://github.com/huljar/prince-vhdl

[110] J. Strömbergson. Siphash verilog implementation. [Online]. Available: https://github.com/secworks/siphash

[111] riscv-tests benchmark repository. [Online]. Available: https://github.com/riscv/riscv-tests

[112] T. Ahmad. Gcm-aes verilog implementation. [Online]. Available: https://opencores.org/projects/gcm-aes

[113] J. Strömbergson. sha256 verilog implementation. [Online]. Available: https://github.com/secworks/sha256